UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Ingeniería en Informática
Proyecto Fin de Carrera

"Manual para el diseño de infraestructuras MMO"

Autor: Miguel Ángel García del Moral
Tutor: Juan Peralta Donate

Agradecimientos

A mi familia en general y **a** mis padres en particular por estar siempre ahí, ofreciendo consejo, apoyo y comprensión, por guiarme en mi educación, y por incontables motivos más, sin vosotros esta meta no hubiera sido alcanzable.

A mi pareja Laura por su estoico aguante a algoritmos y protocolos de comunicación, y por sus ánimos y halagos que me han permitido sentirme capaz de lograr lo que me proponga.

A mi tutor Juan por toda la ayuda prestada, también por las facilidades que me ha ofrecido, guía donde he dudado y perseverancia sin la cual el proyecto aún no habría alcanzado su fin.

Y en general a toda la Universidad Carlos III, ya que han sido sus profesores quienes impartiendo su conocimiento me han formado y motivado en mi interés por esta ingeniería.

Contenido

Tablas

Imágenes

1.Introducción

1.1.Descripción de los capítulos del proyecto

En el presente documento se tratará el problema introducido en el desarrollo de un videojuego multijugador masivo online (MMO), abordando los diferentes frentes que afectan tanto a su diseño como a su posterior implementación.

El documento presenta la siguiente estructura:

- **Introducción**. Presentación del problema y recopilación y definición de términos utilizados para desarrollar el documento.
- **Objetivos del proyecto**. Definición de los objetivos perseguidos en el proyecto y sobre los que trata el documento.
- **Estado de la cuestión**. Estudio del estado actual del mundo del videojuego y su apartado online; introducción a los conceptos necesarios de comunicaciones haciendo uso de redes de ordenadores para crear una base sobre la que resolver el problema del MMO; y presentación de las librerías y motores de juegos para la posterior elección sobre la que se desarrollará la aplicación cliente.
- **Manual de desarrollo de infraestructura MMO**. Presentación de todos los problemas y estudio de alternativas de solución a los que una infraestructura MMO tiene que dar soporte; elección y manual de explicación de solución de implementación; y elección y manual de puesta en marcha de librería o motor de juego elegido para la aplicación cliente.
- **Caso práctico**. Implementación de un caso práctico que presente todos los problemas y soluciones abarcadas en el manual de desarrollo.
- **Conclusiones**. Consideraciones finales alcanzadas una vez planteado el problema, propuesta la solución y vista la respuesta en la práctica.
- **Líneas futuras**. Proposiciones de implementación que no se han tratado o realizado en el caso práctico.
- **Gestión del proyecto**. Presentación gráfica de las fases, tareas y actividades programadas para el desarrollo del proyecto indicando las relaciones entre ellas, recursos asignados y estimaciones de tiempo.
- **Bibliografía**. Colección de fuentes que fueron consultadas para el desarrollo del presente documento.

2.Objetivos del proyecto

El objetivo principal del que se partió para este proyecto fin de carrera ha sido el estudio de la problemática encerrada por el desarrollo de un videojuego multijugador, objetivo que durante la elaboración del presente manual y la implementación de la prueba de concepto evolucionó de forma natural hasta lo que ha sido finalmente, el estudio de la problemática a la que tiene que dar soporte la infraestructura un videojuego multijugador masivo.

De este modo y partiendo de una primera idea en la que se tenían múltiples jugadores conectados en una misma sesión surge la pregunta, ¿cuántos jugadores se podrían soportar con esta solución?, la respuesta a esta pregunta fue otra pregunta, ¿cuántos jugadores se quieren soportar?

Se define así el objetivo final, ¿qué sería necesario para soportar múltiples jugadores de forma masiva en un mismo juego?

Punto a parte del objetivo final, es la motivación personal por encontrar una solución a esta problemática. Por un lado la afición por el mundo de los videojuegos, la inquietud por conocer y participar en el proceso de desarrollo de uno, el descubrimiento de las plataformas de desarrollo de estos, etc. Y por otro lado el interés por el ámbito de la informática de las comunicaciones de redes de ordenadores, el diseño e implementación de distintas arquitecturas, el estudio de diferentes protocolos de comunicación así como de sus ventajas e inconvenientes, etc.

Como resultado de todos estos elementos se ha elaborado para este proyecto fin de carrera un manual en el que se presentan:

- Los fundamentos teóricos en que se sustentan los problemas y soluciones que tiene que enfrentar una infraestructura que de soporte a un videojuego multijugador masivo.
- Un estudio de los diferentes problemas que surgen en este tipo de videojuegos.
- Un análisis de las soluciones posibles a los problemas.

De forma adicional al manual, se presentará también un ejemplo de implementación con una selección de las soluciones más viables y óptimas de entre todas las soluciones a los problemas.

3.Estado de la cuestión

3.1.Resumen del estado actual del entretenimiento digital

El sector de los videojuegos es hoy uno de los más rentables de la industria del entretenimiento generando más dinero que el cine. Ha conseguido revolucionar el ocio y el entretenimiento en el hogar y hoy en día esta revolución sigue su curso, apoyándose en su primera etapa del potencial de la televisión y desarrollándose en la actualidad hacia Internet y las conexiones inalámbricas que llevan en volandas el desarrollo del sector.

La historia de los videojuegos se remonta a 1958 cuando el primer programador de videojuegos, Bill Nighinbottham[1], mostró la versión electrónica de un deporte parecido al tenis en una feria tecnológica en la cual despertó un gran interés aunque Bill no consideró que esto le llevase a ninguna parte y no lo registró. Por lo que el invento no tuvo su reflejo comercial hasta Mayo de 1972, año en que apareció la primera videoconsola llamada Magnavox Odyssey, sistema que permitía jugar al simulador de Hill en la televisión.

Imagen 1. Consola Magnavox Odyssey

Pero el verdadero nacimiento de la industria del videojuego no llegó hasta el año 1972, cuando Nolan Bushnell y Ted Dabney fundaron Atari[2]. Bushnell comprendió que la popularización de los videojuegos llegaría a través de las máquinas recreativas de monedas, y contrató a un programador de Odyssey para desarrollar uno de los juegos más exitosos de la historia aprovechando el error cometido por Nighinbottham, el juego Pong. El éxito de Pong permitió a la compañía lanzar una consola para hogares en 1975, también llamada Pong, de la que creó diferentes versiones.

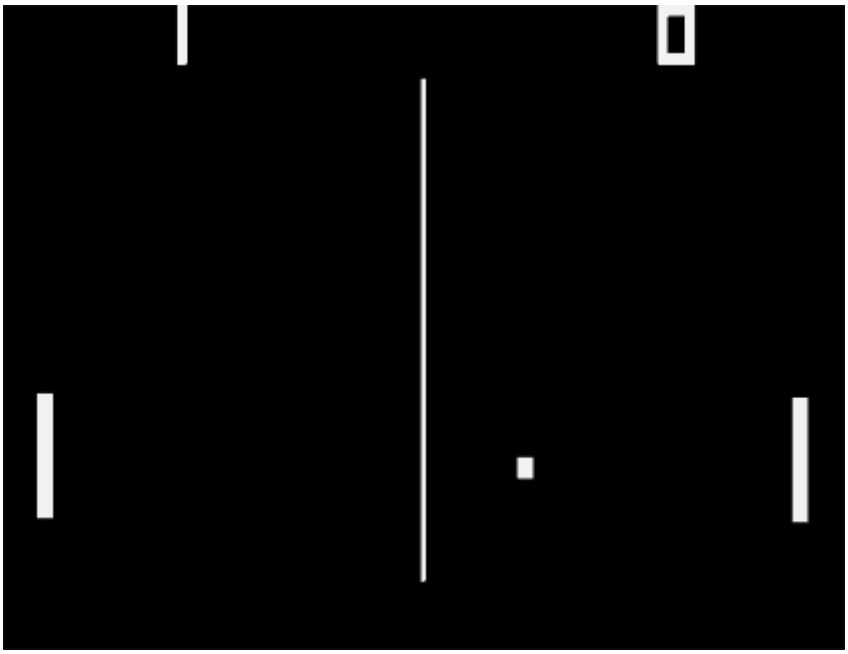

Imagen 2. Captura de pantalla del juego Pong

A finales de los 70 las compañías Magnavox y Atari trabajan en un nuevo sistema que permite jugar a diferentes juegos en un mismo aparato, los resultados fueron la Odyssey 2 y la Atari VCS, posteriormente conocida como Atari 2600 y que permitía la carga de cartuchos distintos. El éxito de Atari con esta nueva consola llegó en los 80 gracias al juego Space Invaders, segundo gran clásico de la historia. El desarrollo de la consola fue magnífico, y con Pac-Man alcanzó su máximo apogeo.

La industria del videojuego prometía y llegaron nuevos competidores provocándose así una gran crisis en el sector que se prolongó hasta el año 1990. Durante la década de los 80 Atari, Mattel, Nintendo[3] y Sega[4] protagonizaron una guerra que generó consolas como la Mattel Intellivision 1980, que supuso una grave amenaza para Atari apoyándose en sonidos más realistas y gráficos de mayor resolución; en un movimiento de contraataque Atari produjo su videoconsola Atari 5200 la cual contaba con el procesador de 8 bits más potente disponible para el hogar; al mercado se sumó la Coleco Colecovision, la cual contaba con la ventaja de poder correr los juegos desarrollados para Atari 2600 y disponía de medios para ampliar el hardware de su sistema; la Sega SG-1000, la cual tras varias revisiones acabaría siendo la conocida Sega Master System en el año 1986, competidora directa de la NES aunque estuviera muy por detrás en ventas respecto a ésta; y finalmente entró en el mercado la videoconsola de 8 bits Nintendo Entertainment System en el año 1987, suponiendo una revolución en todos los aspectos del mundo del videojuego, desde el establecimiento de aspectos primordiales como el diseño del juego y el planteamiento de los controles de éste, hasta un modelo estandarizado referente a la licencia de software para desarrolladores third-party, pasando por la quiebra de empresas especializadas en este sector.

De estas consolas la más exitosa fue la NES de 8 bits, que gracias a su juego Mario Bros alcanzó el liderazgo.

Imagen 3. Videoconsola Nintendo NES

A finales de los 80 Atari en un último intento y Sega adelantándose una vez más a sus competidores revolucionaron y reactivaron el mercado. Sega lanzó en 1989 la MegaDrive, una consola compatible con los juegos de la Master System y que además fue la primera consola de 16 bits reales, doblando así la potencia de NES y su anterior aparato.

Imagen 4. Videoconsola SEGA MegaDrive

NES siguió rivalizando con MegaDrive permitiendo a Sega llegar a dominar el mercado hasta 1992, fecha en la que SuperNintendo llega a Europa y frenó la escalada de Sega aunque no la superara, cosa que si ocurrió en América donde el trabajo conjunto de Nintendo con Squaresoft y la llegada del título Donkey Kong Country contribuyó el liderazgo a la SNES.

Imagen 5. Videoconsola Super Nintendo (SNES)

Atari compró el desarrollo de la compañía Epyx, Handy, la primera consola portátil del mundo. Atari la rebautizó como Lynx, y la consola suponía una de las piezas más avanzadas en tecnología, muy por encima de la NES y casi al nivel de MegaDrive. Sin embargo su elevado precio, su gran tamaño y su elevado consumo energético impidieron su difusión. Sega y Nintendo contraatacaron con las videoconsolas GameGear y GameBoy dominando el mercado portátil y siendo esta última la que mayor éxito presentó.

Imagen 6. Videoconsolas portátiles Sega GameGear y Nintendo GameBoy

La carrera acabó con las consolas de Mattel y Atari que se resignaron al simple desarrollo de juegos. Así pues la lucha quedaba entre Sega y Nintendo, ya que la consola NeoGeo, emuladora de las máquinas recreativas, un sistema que pese a su mayor calidad tuvo impacto en las clases altas debido a su elevado precio.

La llegada de Sony[5] al mercado lo revitalizó, provocando que Sega lanzara la Saturn en 1995, justo antes de la aparición de la PlayStation, ambas consolas de 32 bits. Frente al exitoso comienzo de la Saturn, Sony fue consolidando la posición de PlayStation hasta deshacerse de su competidor Sega. Nintendo reaccionó un año más tarde con la Nintendo 64, que suponía

una mejora del sistema de desarrollo gráfico a 64 bits, pero cometió error de seguir apostando por los cartuchos en lugar de por un soporte en CD-ROM.

Sega volvió a adelantarse al mercado lanzando en 1999 la Dreamcast, la consola más avanzada del mercado. Pero los rumores de un nuevo desarrollo de Sony y el fracaso que tuvo la Saturn acabaron por sepultar la nueva innovación de Sega que anunció su retirada del mercado de las consolas. Casi dos años tardó Sony en lanzar su PS2, cuyo éxito ha sido notable, pese al lanzamiento en 2001 de Nintendo GameCube y Microsoft Xbox.

La guerra por el mercado de las videoconsolas sigue abierta, y las tres empresas tienen ya en el mercado sus nuevas consolas, la Xbox 360 lanzada en Noviembre de 2005, la PS3 lanzada en Noviembre de 2006 y la Nintendo Wii lanzada entre los meses de Noviembre y Diciembre de 2006.

Casi 40 años de evolución para llegar a estas centrales de entretenimiento digital en las que se aúna la tecnología audiovisual, los videojuegos e Internet.

3.1.1.Evolución del multijugador

La experiencia de juego ofrecida por el modo multijugador ha mostrado una importante evolución y un auténtico filón a lo largo de la historia del videojuego.

Nace en 1969 cuando el primer videojuego online creado por Rick Blomme, permitiendo a dos jugadores compartir una misma experiencia de juego sobre el sistema PLATO[6] que supuso uno de los pilares básicos para la creación de las comunidades online.
Este videojuego era una versión del Spacewar desarrollado por Steve Russell y otros estudiantes del MIT (Instituto de Tecnología de Massachusets).

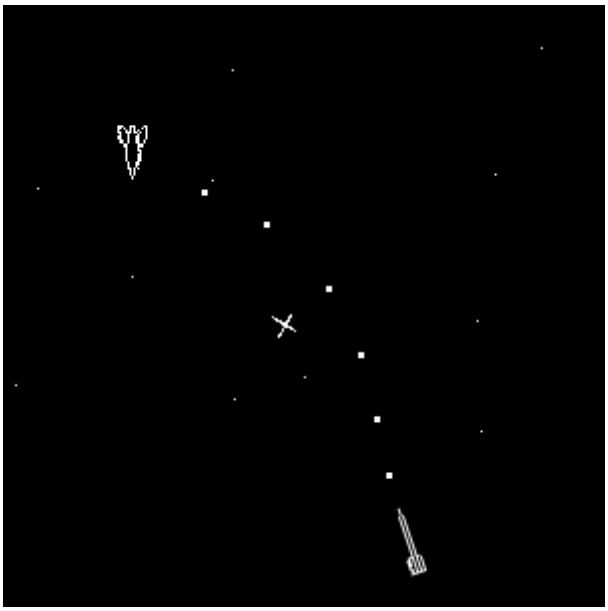

Imagen 7. Captura de pantalla del primer juego multijugador online

El videojuego desarrollado por Rick se basaba en un Sistema de Tablón de Anuncios (conocido como BBS, Bulletin Board System[7]) en el que los movimientos de los jugadores se enviaban al administrador del sistema BBS, pero dada la tecnología de entonces y los movimientos realizados de forma simultánea hacían que los estados de juego observados por los jugadores no estuvieran actualizados con la última información y creando estados inconsistentes del juego en los que los jugadores tomaban decisiones sobre situaciones que en ese momento ya podían haber terminado.
Este tipo de juegos basados en BBS evolucionaron partiendo de que inicialmente los enemigos manejados por la máquina eran en realidad manejados por el administrador del sistema, hasta la automatización de la lógica de estos procesos, de modo que el administrador tenía más libertad para la creación del juego.

Partiendo de estos sistemas, no pasó mucho tiempo hasta que aparecieron los primeros juegos de aventuras basados en texto, a estos juegos se les conocía como MUDs (Multi-user dungeons), que tomaron como base los juegos RPG de papel y lápiz donde el rol del director de juego era llevado por el administrador del sistema BBS permitiendo interactuar con el sistema a jugadores y administrador del sistema a través de un sistema de turnos.

El primero videojuego MUD apareció en 1979, fue diseñado por Roy Trubshaw y Richard Bartle y alojado en el Reino Unido en la Universidad de Essex. Permitía a los jugadores loguearse en el juego y moverse por el mundo, interactuar con otros jugadores y elementos controlados de forma automática mediante codificación en el programa implementando sencillas rutinas de inteligencia artificial, además del uso ítems y un sistema de inventario creando una sensación de realismo que antes no había sido vista, y todo ello a través del uso de comandos de texto. Con todos estos elementos desarrollados se dio paso al siguiente punto de la evolución del multijugador, los juegos de tiros en primera persona conocidos como FPS (First Person Shooter).

Los juegos de tipo FPS cogen su nombre debido a que la cámara se sitúa en primera persona como si el jugador fuera el personaje del videojuego facilitando la visión del entorno y creando una sensación más realista de lo que una experiencia así supone.

El primer juego de tipo FPS en 3D se llamó A-maze-ing y salió para plataforma Macintosh en 1989 y un tiempo después se publicó el que ha sido el verdadero estandarte de este tipo de juegos, Doom, desarrollado por ID Software en 1994 y que poco tiempo después continuó la serie con Doom II. Doom ya tenía la capacidad de soportar en la misma partida hasta 4 jugadores.

Animado por el éxito de la serie Doom, ID Software desarrolló el videojuego Quake para ampliar la experiencia multijugador dando soporte hasta a 16 jugadores.

En este punto y de forma paralela a los FPS y vista la aceptación de los videojuegos multijugador tipo MUD, surgen los primeros juegos de estrategia basados en turnos y los juegos basados en navegadores Web como Archmage y Terranlegacy, en los que jugadores interactuaban teniendo limitaciones de turnos y comunicación a través del navegador Web con otros jugadores.

De los juegos de estrategia basados en turno, se dio el salto a los juegos de estrategia en tiempo real. La encargada de colocar este primer pilar fue Blizzard en 1994 con su videojuego Warcraft: Orcs & Humans que permitía compartir una misma partida entre dos usuarios a través de un multijugador competitivo.

Mientras tanto, aunque el primer MMP (massively multiplayer – multijugador masivo) ya había sido desarrollado en 1987 con el nombre de Air Warrior, la llegada de Ultima Online en 1997 supuso la verdadera entrada de este tipo de juegos al mundo multijugador.

En Ultima Online se proponía al jugador una experiencia de videojuego de rol multijugador masivo en línea en un mundo de fantasía en el que el jugador podía desarrollar habilidades, estadísticas e incluso un sistema de artesanía que dio paso a un sistema de economía en el que los personajes manejados por la máquina (NPCs – Non player caracters, personajes no jugadores) compraban objetos de los jugadores basados en una simulación de oferta y demanda.

Siguiendo el camino empezado por Ultima Online, la compañía Verant Interactive desarrolló Everquest, otro juego que siendo multijugador masivo en línea seguía la línea de los

RPG en un mundo de fantasía en el que los jugadores interactuaban unos con otros, obtenían experiencia, subían de niveles para conseguir nuevas habilidades, etc.

Finalmente y como evolución natural de lo que los juegos multijugadores masivos aportaban en común, un mundo virtual compartido por todos los jugadores, aparece en 2003 Second Life desarrollado por Linden Labs.

En este tipo de mundo virtual lo que se trata de recrear es la experiencia de la vida real aplicada a un mundo virtual donde los jugadores pueden interactuar unos con otros, crear negocios, comprar bienes virtuales, crear sus propias casas, como si del Metaverso de Neal Stephenson[8] se tratara.

3.2.Comunicaciones

3.2.1.Redes de ordenadores

Una red de ordenadores es el conjunto de dos o más ordenadores conectados entre sí por medio de cables, señales, o cualquier otro método de transporte de datos, y que comparten información, programas o ambos.

Estas redes pueden ser clasificadas por distintos criterios:

- **Alcance**: Red de área personal (PAN), red de área local (LAN), red de área metropolitana (MAN), red de área amplia (WAN).
- **Método de conexión**: medios guiados como el cable coaxial, par trenzado o fibra óptica; medios no guiados como radio, infrarrojos, microondas o láser.
- **Relación funcional**: Cliente-servidor, Par a par (p2p, peer to peer).
- **Topología de red**: Red en bus, estrella, en anillo, en malla, árbol o mixta.
- **Direccionalidad de los datos**: Simplex, Half-duplex y Duplex.

Para lograr el enlace entre las computadoras y los medios de transmisión es necesaria la intervención de una tarjeta de red o NIC con la cual se puedan enviar y recibir paquetes de datos desde y hacia otras computadoras. Para ello se empleará un protocolo que permita la comunicación y convierta esos datos a un formato que pueda ser transmitido por el medio.

Cabe señalar que a cada tarjeta de red le es asignado un identificador único por su fabricante, conocido como dirección MAC, que consta de 48 bits. Dicho identificador permite direccionar el tráfico de datos de la red del emisor al receptor adecuados.

3.2.2.Sistemas distribuidos

Se definen como “Colección de computadoras separados físicamente y conectados entre sí por una red de comunicaciones distribuida; cada máquina posee sus componentes de hardware y software que el usuario percibe como un solo sistema”[9].

Los sistemas distribuidos deben ser muy confiables ya que si un componente del sistema presenta un fallo, otro componente debe de ser capaz de reemplazarlo, denominándose este ámbito Tolerancia a Fallos[10].

Las características que presentan estos sistemas son las siguientes:

- **Recursos compartidos.** Los recursos en un sistema distribuido están físicamente encapsulados en una de las computadoras y sólo pueden ser accedidos por otras computadoras mediante las comunicaciones, y estos recursos pueden ser de tipo físico como componentes hardware (unidades de disco duro, impresoras, etc.); y de tipo lógico como componentes software (ficheros, bases de datos, etc.).
- **Concurrencia.** Cuando existen varios procesos en una única maquina decimos que se están ejecutando concurrentemente. Esta característica se puede dar por dos razones: cuando múltiples clientes quieren acceder a un mismo recurso, y cuando en el servidor se ejecutan de forma concurrente varias aplicaciones. Este apartado se puede ver ampliado en el punto 3.2.8 Concurrencia.
- **Escalabilidad.** Los sistemas distribuidos deben operar de manera efectiva y eficiente a diferentes escalas, por lo que el software de sistema como el de aplicación no deberían cambiar cuando la escala del sistema se incrementa. Esta necesidad de escalabilidad no es solo un problema de prestaciones de red o de hardware, sino que está íntimamente ligada con todos los aspectos del diseño de los sistemas distribuidos. El diseño del sistema debe reconocer explícitamente la necesidad de escalabilidad o de lo contrario aparecerán serias limitaciones.
- **Tolerancia a fallos.** Cuando se producen fallos en el software o en el hardware los programas podrían producir resultados incorrectos o podrían pararse antes de terminar la computación que estaban realizando. El diseño de sistemas tolerantes a fallos se basa en dos cuestiones, complementarias entre sí: Redundancia hardware (uso de componentes redundantes) y recuperación del software (diseño de programas que sean capaces de recuperarse de los fallos).
- **Transparencia.** La transparencia se define como la ocultación al usuario y al programador de aplicaciones de la separación de los componentes de un sistema distribuido, de manera que el sistema se percibe como un todo, en vez de una colección de componentes independientes. La transparencia ejerce una gran influencia en el diseño del software de sistema.

3.2.3.Modelo OSI

El modelo de referencia OSI (Interconexión de Sistemas Abiertos) fue creado por la Organización Internacional para la Estandarización[11] como un marco de referencia para la definición de arquitecturas de sistemas de comunicaciones y salvar el problema que suponía en la comunicación las diferencias que presentan equipos de cómputo distintos (procesador, interfaces de comunicación, sistema operativo, etc.). Por lo que el modelo en sí no define un estándar de comunicaciones, pero sí que ha servido para que estándares y protocolos se fundamenten en una misma idea.

El modelo OSI se diseñó con una estructura multinivel, de modo que cada capa resolviera una parte del problema de comunicación. Las capas o niveles, son los siguientes:

Imagen 8. Estructura de capas del modelo OSI

El objetivo de cada una de estas capas es el siguiente:

Capa física

Su función es transformar las tramas de datos recibidas por la capa de enlace y transmitirlas a nivel de bit ocupándose de las propiedades físicas y características eléctricas de los componentes.

Define el medio de comunicación utilizado para la transferencia de la información que puede ser un medio guiado como el cable de par trenzado y la fibra óptica; o un medio no guiado como infrarrojos y redes inalámbricas.

Dispone del control de este medio a través de características como el tipo de cable y calidad del mismo y la forma de transmitir la información como la codificación de la señal y modulación.

Algunos ejemplos de protocolos de este nivel son: IEEE 802.11x, Bluetooth, IrDA y V.92.

Capa de enlace

Esta capa debe ocuparse del direccionamiento físico, topología de la red, acceso a la red, notificación de errores, distribución ordenada de tramas y control del flujo. Consta de dos subcapas: subcapa LLC (control del enlace lógico) que da soporte a servicios orientados y no orientados a conexión administrando las comunicaciones entre dispositivos; y subcapa MAC (control de acceso al medio) que administra el protocolo de acceso al medio físico de red y permite que varios dispositivos se identifiquen de forma unívoca.

Cualquier medio de transmisión debe ser capaz de proporcionar un tránsito de datos fiable, sin errores, a través de un enlace físico.

Debe crear y reconocer los límites de las tramas organizando 1's y 0's en grupos lógicos y viceversa, así como resolver los problemas derivados del deterioro, pérdida o duplicidad de las tramas.

Adicionalmente puede incluir algún mecanismo de regulación del tráfico evitando así la saturación de un receptor que sea más lento que el emisor.

Algunos ejemplos de protocolos de este nivel son: Ethernet, Fast Ethernet, Gigabit Ethernet y Token Ring.

Capa de red

En este nivel se realiza el direccionamiento lógico y la determinación de la ruta de los datos desde su origen hasta el destino, aun cuando ambos no estén conectados directamente definiendo el enrutamiento y envío de paquetes entre redes. La unidad de datos de esta capa es el paquete.

También controla que no se produzca congestión de la red, que es el fenómeno que se produce cuando se satura un nodo de la red.

Algunos ejemplos de protocolos de este nivel son: IP, ARP, RARP e ICMP

Capa de transporte

Capa encargada de efectuar el transporte de los datos de la máquina origen a la de destino, independizándolo del tipo de red física que se esté utilizando. La unidad de datos de esta capa es el segmento.

Su función básica es aceptar los datos enviados por las capas superiores, dividirlos en pequeñas partes si es necesario, y pasarlos a la capa de red por lo que debe aislar a las capas superiores de las distintas posibles implementaciones de tecnologías de red en las capas inferiores, convirtiéndola en el centro de la comunicación de este modelo.

Todo el servicio que presta la capa está gestionado por las cabeceras que agrega al paquete a transmitir.

Algunos ejemplos de protocolos de este nivel son: TCP, UDP y SPX.

Capa de sesión

Esta capa es la que se encarga de mantener el enlace entre los dos computadores que estén transmitiendo datos.

Ofrece los siguientes servicios: control de la sesión entre emisor y receptor conociendo quién emite y quién recibe; control de la concurrencia para el acceso a la misma operación crítica; y mantener puntos de verificación para que interrumpida la conexión se pueda retomar desde el último punto de verificación.

Algunos ejemplos de protocolo de este nivel son: SSL, NetBIOS y RPC.

Capa de presentación

Esta capa es la encargada de manejar la estructura de datos abstracta y realizar las conversiones de representación de los datos necesarios para la correcta interpretación de los mismos cuando la comunicación se establece entre máquinas diferentes.

Se puede decir que hace las veces de traductor, permitiendo cifrar los datos y comprimirlos adicionalmente.

Un ejemplo de protocolo de este nivel es ASN1.

Capa de aplicación

Ofrece a las aplicaciones la posibilidad de acceder a los servicios de las demás capas y define los protocolos que utilizan las aplicaciones para intercambiar datos.

El usuario final interactúa sobre aplicaciones que ofrecen una interfaz sobre este nivel.

En este nivel no hay un número determinado de protocolos, ya que cada aplicación puede definir el suyo propio, pero entre los más conocidos encontramos HTTP, SMTP, FTP, POP, SSH, DNS, entre otros.

3.2.4.Protocolos de nivel de transporte

- **Protocolo UDP**

UDP es un protocolo estándar[12] que permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión entre las máquinas.

De este modo el único requisito necesario para la comunicación de información consiste en que a la aplicación que envía información (cliente) se le indique la dirección IP y el puerto al que se dirige, y en la aplicación que recibe la información (servidor) se especifique por qué puerto se quiere escuchar.

- Ventajas:
  - El datagrama contiene toda la información necesaria para llegar al destino, por lo que **no es necesario establecer una conexión previa** con el otro extremo
  - No introduce retardos en el envío de información ya que no ofrece garantía en la recepción de la información
- Inconvenientes:
  - **No tiene confirmación** de recepción, por lo que no se garantiza que al enviar un datagrama éste llegue a su destino
  - **No tiene control de flujo**, por lo que los datagramas pueden llegar al otro extremo en distinto orden al enviado.

De estas características podemos deducir que este protocolo es ideal cuando la velocidad de transmisión es más importante que la seguridad de que llega la información debido al retardo que introduce este control, o cuando se quiere comunicar información a muchas máquinas y es necesario ahorrar el tiempo de establecimientos de conexión.

Un ejemplo de aplicación de este protocolo es el streaming de audio y video.

Por último es interesante mencionar que en determinadas aplicaciones es necesaria la velocidad de UDP y la garantía de recepción de la información así como su ordenación.

Dedicados a este aspecto hay grupos de desarrollo como por ejemplo el proyecto ENET[13] que han trabajado en la creación de librerías que envuelven el protocolo de comunicaciones UDP y que ofrecen un comportamiento similar al de TCP, consiguiendo mejorar el rendimiento para casos muy concretos.

- **Protocolo TCP**

TCP es un protocolo que permite el envío de paquetes a través de la red previo establecimiento de conexión.

- Ventajas:
  - Ofrece un servicio fiable, por lo que se tiene la garantía de que los paquetes llegarán al otro extremo
  - Tiene control de flujo, por lo que los paquetes llegarán al otro extremo en el mismo orden que fueron enviados.
- Inconvenientes:
  - Cada vez que se quiere comunicar dos máquinas es necesario establecer primero la conexión, y una vez se ha terminado de enviar la información se tiene que cerrar esta conexión.

Para que el protocolo TCP ofrezca este comportamiento, la aplicación que lo implemente sigue el siguiente ciclo de ejecución en la comunicación:

1. **Establecimiento de conexión**. Creación del socket y conexión con el extremo al que se quiere comunicar la información. Produce el llamado Three-Way Handshake a través del cual se establece la conexión entre cliente y servidor

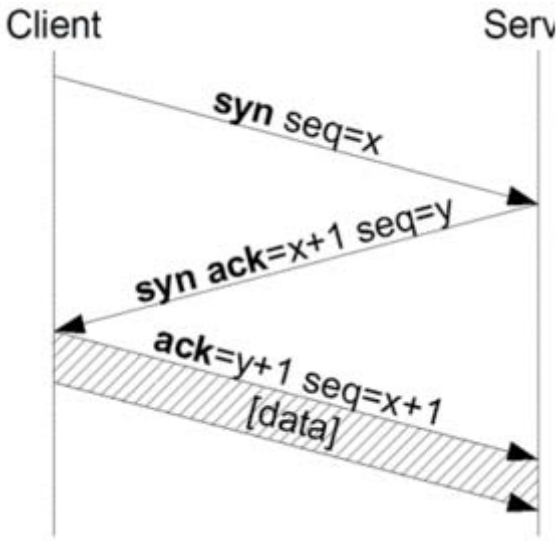

Imagen 9. Establecimiento de conexión bajo protocolo TCP/IP. Three-way handshake

2. **Envío de información de un extremo a otro.** Una vez se ha establecido la conexión entre cliente y servidor se realiza la transferencia de datos.

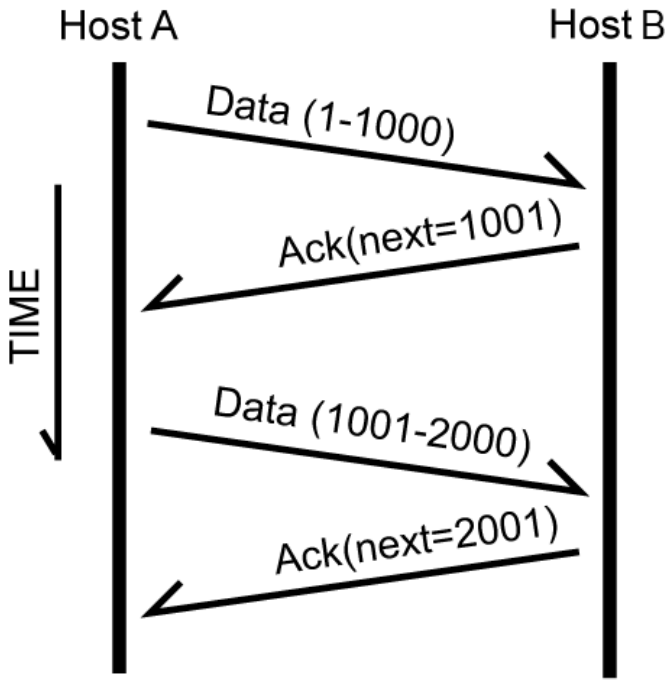

Imagen 10. Comunicación de datos bajo protocolo TCP/IP.

3. **Cierre de conexión**. Finalizado el envío de datos, es necesario cerrar la conexión para finalizar el ciclo de comunicación bajo el protocolo TCP

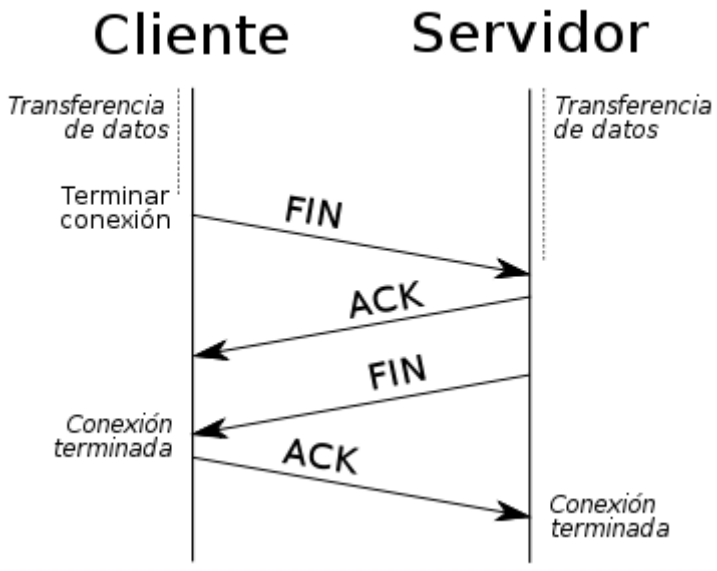

Imagen 11. Cierre de conexión bajo protocolo TCP/IP

De estas características se puede deducir que este protocolo es ideal cuando se necesita la garantía de que la información llegue al otro extremo, siempre teniendo en cuenta

el coste de establecimiento de conexión, el retardo que se puede introducir en la recepción del paquete en caso de pérdida y el cierre de conexión.

Algunos ejemplos de utilización de este protocolo es la navegación por la Web.

3.2.5.Modelo TCP/IP

El modelo TCP/IP no se origina en un comité de estándares, sino que proviene de las investigaciones realizadas por DARPA[14] respecto al conjunto de protocolos de TCP/IP.

Cada nivel del modelo TCP/IP corresponde a uno o más niveles del modelo de referencia OSI, está estratificado por capas de modo que la información que se envía circula desde la capa más próxima al usuario hasta la más cercana al hardware, añadiendo cabeceras con información relativa a cada capa en su paso por ellas. Esta estratificación se divide según el siguiente diagrama:

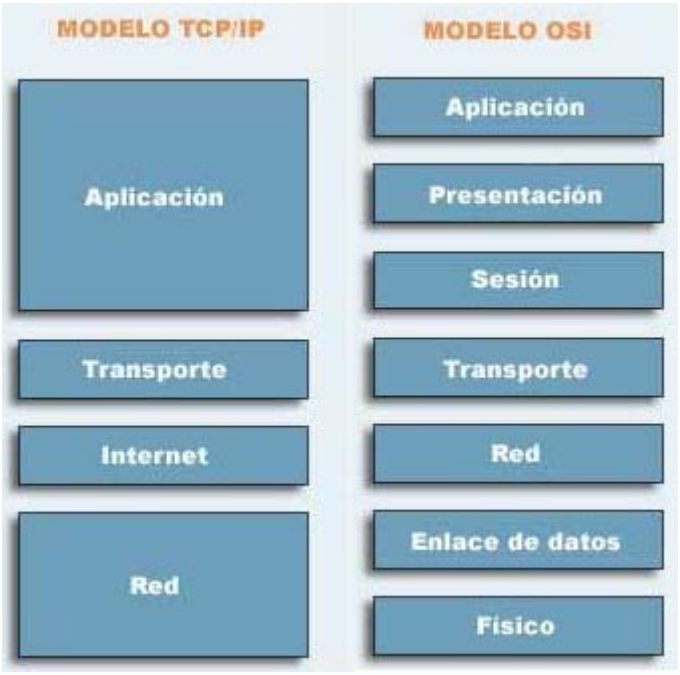

Imagen 12. Relación de capas entre los modelos TCP/IP y OSI

El objetivo de cada una de estas capas es el siguiente:

Capa de red

Especifica información detallada de cómo se envían físicamente los datos a través de la red, que incluye cómo se realiza la señalización eléctrica de los bits mediante los dispositivos de hardware que conectan directamente con los medios de red guiados y no guiados.

Algunos protocolos pertenecientes a esta capa son: Ethernet, Token Ring, y X.25.

Capa de Internet

Empaqueta los datos en datagramas IP que contienen información de las direcciones de origen y destino utilizada para reenviar los datagramas entre hosts y a través de redes. Realiza el enrutamiento de los datagramas IP.

Algunos protocolos pertenecientes a esta capa son: IP, ICMP, ARP y RARP.

Capa de transporte

Permite administrar las sesiones de comunicación entre equipos host y define el nivel de servicio y el estado de la conexión utilizada al transportar datos.

Algunos protocolos pertenecientes a esta capa son: TCP, UDP y RTP.

Capa de aplicación

Define los protocolos de aplicación TCP/IP y cómo se conectan los programas de host a los servicios del nivel de transporte para utilizar la red.

Algunos protocolos pertenecientes a esta capa son: HTTP, Telnet, FTP, TFTP, SNMP, DNS y SMTP.

3.2.6. Nociones básicas de comunicación a través de la red IP

El protocolo IP, situado en la pila de protocolos a nivel de la capa de red, permite la comunicación de datos desde un origen a un destino.

Para establecer esta comunicación desde un origen a un destino es necesario contar con los siguientes elementos: dirección IP de la máquina en la que está corriendo la aplicación servidora con la que se quiere conectar, el puerto por el que ofrece servicio y el protocolo que utiliza para comunicarse.

Conocidos estos datos el origen podrá enviar y recibir información indicando que se quiere comunicar con la máquina destino que tiene la IP **X.X.X.X** en el caso de IPv4 o la IP **X:X:X:X:X:X:X:X** en el caso de IPv6, por el puerto **Y**, y enviando la información usando el protocolo TCP o UDP.

En lo que respecta a la dirección IP, se trata de un elemento necesario ya que identifica de forma unívoca una máquina en la red, de modo que sin conocer esta dirección no se podrá establecer la comunicación ya que sería imposible localizarla.

Para el caso de IPv4 estará compuesta de 4 grupos de números naturales dentro del intervalo [0,255] divididos entre ellos con el carácter punto, siendo ejemplos válidos 192.168.0.1, 80.125.100.5 o 57.57.102.2.

En el caso de IPv6 estará compuesta de 8 grupos de 4 números en base hexadecimal separados entre ellos con el carácter dos puntos, siendo ejemplos válidos 2001:0DB8:0000:0000:0000:0000:1428:57ab o 2001:0db8:85a3:0000:1319:8a2e:0370:7344.

Respecto a los protocolos, tanto TCP como UDP tienen 65536 puertos (2^16 puertos independientes) por los que establecer comunicación, esto quiere decir que el puerto 1 de TCP no es el mismo que el puerto 1 de UDP y en una misma máquina se podría mantener la comunicación por los dos a la vez sin que esto presentara un conflicto.

En cuanto al puerto, sirve para identificar en el destino la aplicación a la que se enviará la información.

Todos estos datos en conjunto permiten tener en una misma máquina distintas aplicaciones ejecutándose a la vez y comunicándose por distintos puertos y/o protocolos.

3.2.7.Tipos de arquitecturas de red en sistemas distribuidos

La arquitectura de red es el medio para desarrollar e implementar un conjunto coordinado de servicios que se puedan interconectar. Define el plan según el cual se conectan los protocolos y otros programas de software.

Las arquitecturas más conocidas en sistemas distribuidos son las siguientes:

- **Arquitectura cliente-servidor**

Los equipos que participan en este tipo de arquitecturas se definen dentro de los dos roles que se proponen: cliente que realizará peticiones, y servidor que dará respuesta a estas peticiones. Esta separación es de tipo lógico por lo que no se ejecutará necesariamente cada uno de estos elementos en una sola máquina o en un solo programa.

Las características que cumple un **cliente** son:

- Realiza las peticiones y por tanto tiene un papel activo en la comunicación.
- Espera y recibe las respuestas del servidor.

Las características que cumple un **servidor** son:

- Al iniciarse esperan a que lleguen las solicitudes de los clientes, desempeñan entonces un papel pasivo en la comunicación.
- Tras la recepción de una solicitud o petición, la procesan y luego envían la respuesta al cliente.
- Por lo general, aceptan conexiones desde un gran número de clientes replicándose a sí mismo en múltiples hilos o procesos (este número puede estar limitado).

Las ventajas que introduce este modelo son:

- **Centralización del control**. La integridad de la información puede ser controlada por el servidor, de modo que un cliente defectuoso no puede alterar el sistema, facilitando también la actualización de información en el resto de clientes dado lo sencillo de su estructura.
- **Escalabilidad**. Se puede aumentar de forma independiente el número de clientes y servidores.
- **Mantenimiento sencillo**. Dado que se pueden distribuir las funciones entre varios ordenadores independientes, es más fácil el reemplazo, reparación y movimiento de servidores.

Los inconvenientes que presenta este modelo son:

- **Congestión de tráfico**. Ya que el servidor gestiona todas las peticiones de los clientes, si no se hace un buen balance este se puede ver saturado.
- **Falta de robustez**. Al canalizarse la información a un servidor, si este no está disponible las peticiones de los clientes pueden perderse.
- **Hardware y software específico**. Para satisfacer las peticiones de múltiples clientes y en función del servicio proporcionado por el servidor, el servidor puede verse en la necesidad de un hardware que ofrezca un mejor rendimiento con su consecuente repercusión en el coste.

Por como atiende el servidor a las peticiones de los clientes puede ser implementado para ser concurrente o no.

En el caso de un servidor no concurrente se dará servicio a un único cliente en un mismo instante de tiempo y hasta que no se procese y se responda si se precisa al cliente, no se atenderá al siguiente cliente. Un gráfico del funcionamiento de este tipo de implementación cliente-servidor es el siguiente:

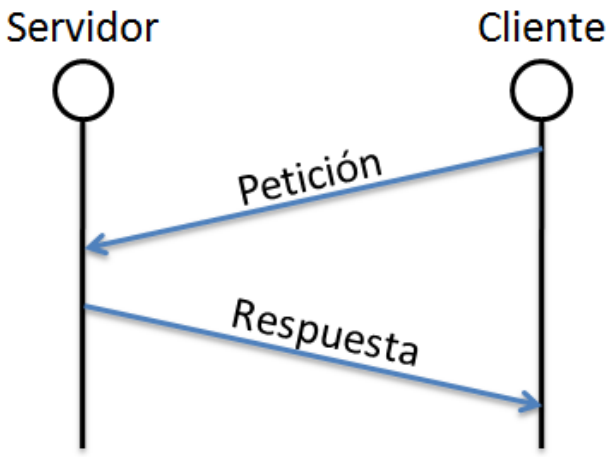

Imagen 13. Esquema de arquitectura cliente-servidor no concurrente

En el caso de un servidor concurrente se podrá dar servicio a varios clientes al mismo tiempo sin necesidad de que el servidor haya terminado de procesar la petición de otro cliente, para ello al recibir la conexión de un cliente se creará un proceso o proceso ligero que se encargará de atender la petición del cliente que establece la conexión. Un gráfico de esta implementación de la arquitectura es el siguiente:

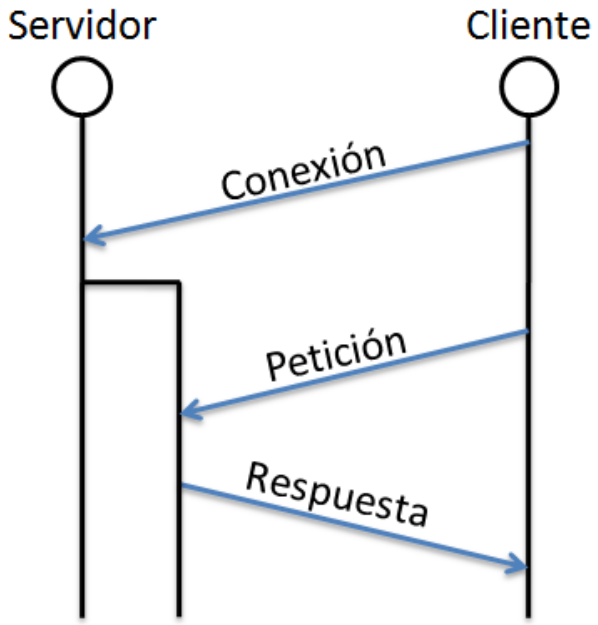

Imagen 14. Esquema de arquitectura cliente-servidor concurrente

- **Arquitectura par a par**

En las arquitecturas par a par (o *P2P*) todos los equipos que participan se definen como nodos iguales, realizando tanto el rol de servidor como de cliente. Estas redes permiten compartir e intercambiar información entre todos los nodos, lo cual ha generado mucha polémica debido a su uso para intercambiar ficheros sujetos a leyes de Copyright[15].

Las redes par a par aprovechan, administran y optimizan el uso del ancho de banda de los demás usuarios de la red por medio de la conectividad entre los mismos, obteniendo más rendimiento en las conexiones y transferencias que con algunos métodos centralizados convencionales, donde una cantidad relativamente pequeña de servidores provee el total del ancho de banda y recursos compartidos para un servicio o aplicación.

Las ventajas que presenta esta arquitectura son:

- **Escalabilidad**. Estas redes cuentan con millones de usuarios, siendo lo deseable que a mayor número de usuarios mejor rendimiento debido al incremento de recursos totales del sistema.
- **Robustez**. La naturaleza distribuida de las redes par a par también incrementa la robustez en caso de haber fallos en la réplica excesiva de los datos hacia múltiples destinos. Esta robustez se ve incrementada al carecer de un nodo que ejerza de servidor centralizado.
- **Descentralización.** Ya que todos los nodos son iguales, no hay ninguno que realice ninguna función especial, por lo que no es necesario tener una máquina con hardware específico en la red.
- **Balanceo de carga.** Los costes están repartidos entre todos los usuarios de la red.

Las desventajas que presenta esta arquitectura son:

- **Localización**. Un problema que se tiene que resolver en este tipo de arquitecturas es la localización de un nodo conectado en la red. Esta desventaja se suele resolver rompiendo el esquema presentado por la arquitectura e introduciendo un servidor que sea conocido por todos los nodos y que mantenga una lista con las direcciones de los nodos conectados.
- **Seguridad**. Debido a que la información puede usar nodos intermedios que no son destinatarios para hacerla llegar hasta su nodo final, es necesario implementar algún sistema de seguridad para evitar el espionaje de las comunicaciones entre los nodos.

Adicionalmente, estas arquitecturas se pueden clasificar según los siguientes grupos:

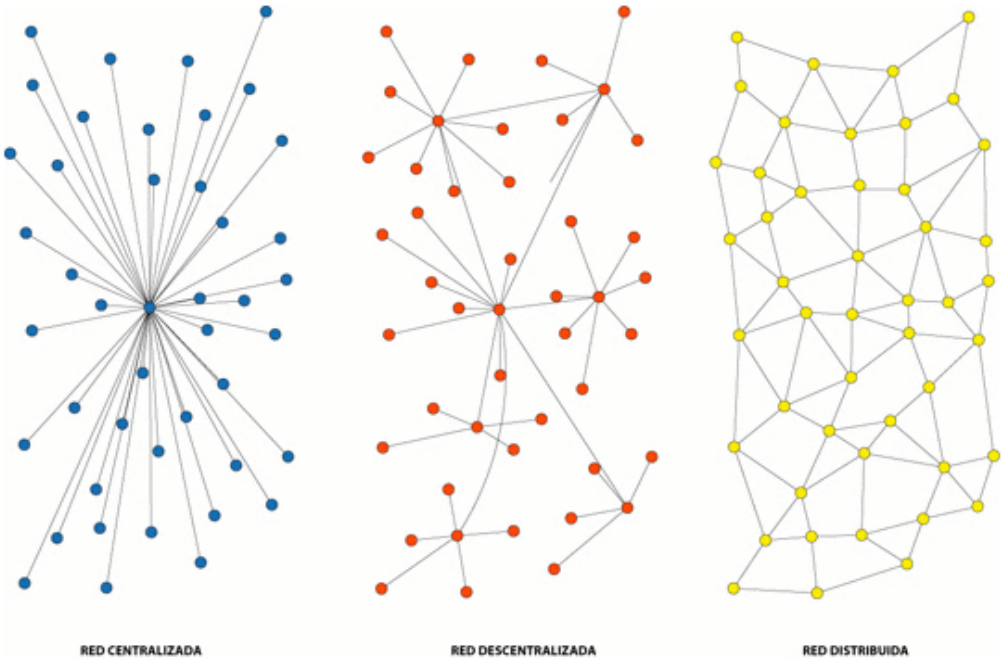

Imagen 15. Clasificación de redes P2P

- *Red centralizada.*

Este tipo de red se basa en una arquitectura monolítica en la que todas las transacciones se hacen a través de un único servidor que sirve de punto de enlace entre dos nodos y que, a la vez, almacena y distribuye los nodos donde se almacenan los contenidos.

Poseen una administración muy dinámica y una disposición más permanente de contenido. Sin embargo, está muy limitada en la privacidad de los usuarios y en la falta de escalabilidad de un sólo servidor, además de ofrecer problemas en puntos únicos de fallo, situaciones legales y enormes costos en el mantenimiento así como el consumo de ancho de banda.

- *Red descentralizada*

En este tipo de red se puede observar la interacción entre un servidor central que sirve como HUB y administra los recursos de banda ancha, enrutamientos y comunicación entre nodos pero sin saber la identidad de cada nodo y sin almacenar información alguna, por lo que el servidor no comparte archivos de ningún tipo a ningún nodo.

Tiene la peculiaridad de ambas maneras, puede incorporar más de un servidor que gestione los recursos compartidos, pero también en caso de que el o los servidores que gestionan todo caigan, el grupo de nodos sigue en contacto a través de una conexión directa entre ellos mismos con lo que es posible seguir compartiendo y descargando más información en ausencia de los servidores

- *Red distribuida.*

  Las redes de este tipo son las más comunes siendo las más versátiles al no requerir de una gestión central de ningún tipo, lo que permite una reducción de la necesidad de usar un servidor central, por lo que se opta por los mismos usuarios como nodos de esas conexiones y también como almacenistas de esa información. De este modo todas las comunicaciones son directamente de usuario a usuario con ayuda de un nodo (que es otro usuario) quien permite enlazar esas comunicaciones.

- **Arquitectura N-Tiered**

El modelo cliente-servidor asume exactamente dos participantes discretos en el sistema, por ello se denomina sistema 'two-tier'; la lógica de la aplicación puede estar en el cliente, el servidor, o compartida entre los dos.

En los sistemas N-Tiered se tiene una subdivisión del conjunto del sistema en tres o más capas. Por ejemplo, si la lógica de la aplicación se ve separada de los datos y de la interfaz de usuario, el sistema se convierte en 'three-tiered'. En un sistema 'three-tiered' ideal toda la lógica de la aplicación reside en una capa separada.

- **Arquitectura Clúster**

Es un grupo de múltiples ordenadores unidos mediante una red de tal forma que el conjunto es visto como un único ordenador. Los clústeres son usualmente empleados para mejorar el rendimiento y/o la disponibilidad por encima de la que es provista por un solo ordenador siendo más económico que una máquina individual de rapidez y disponibilidad comparable.

Los clústeres ofrecen las siguientes características:

- **Alto rendimiento**. Ofrecen gran capacidad computacional, gran cantidad de memoria, o ambas a la vez.
- **Alta disponibilidad**. Ofrecen un servicio confiable con detección de fallos y recuperación ante estos.
- **Alta eficiencia**. Ofrecen un servicio en el que prima la capacidad de ejecución de un mayor número de tareas en el menor tiempo posible.
- **Escalabilidad**. Capacidad para ampliar la red de ordenadores que componen el clúster sin que este se vea afectado por ello.

La tecnología de clústeres ha evolucionado en apoyo de actividades que van desde aplicaciones orientadas a la supercomputación, servidores Web, soporte a infraestructuras de comercio electrónico, bases de datos de alto rendimiento, etc, marcando unos objetivos en según qué ámbito se aplique el clúster, siendo por ejemplo más importante la alta disponibilidad y eficiencia en casos de clústeres para sistemas comerciales, y de alto rendimiento aquellos orientados a estudios científicos.

Desde una perspectiva general, los elementos software y hardware que participan en los clústeres son:

- **Nodos**. Las máquinas que componen la red que conforma el clúster, pueden ser ordenadores, sistemas multi-procesador, etc. Es deseable que sean de características lo más similares posibles, ya que sino al asignar el middleware el trabajo pueden producirse un mal balanceo de la carga.
- **Sistemas operativos**. Deben ser multiprocesador y es deseable que también sean multiusuario. Sobre ellos correrá el middleware.
- **Conexiones de red**. A través de estas se verán interconectado los distintos nodos del clúster. Para establecer un clúster basta con una simple red Ethernet, pero dependiendo del objetivo perseguido en el clúster puede ser necesario el uso de redes de alta velocidad para minimizar la latencia y obtener un mayor ancho de banda.
- **Middleware**. Es el software que encontrándose entre el sistema operativo y las aplicaciones se encarga de generar la sensación de que cada máquina individual trabaje como un único ordenador más potente, recibiendo los trabajos a distribuir en el clúster y gestionándolos para una rápida ejecución, cubriendo además las necesidades del balanceo de carga que pueda surgir de la sobrecarga en alguno de los nodos; mantenimiento de servidores de modo que si un servidor necesita mantenimiento o actualización se migren los procesos a otro nodo; y priorización de los trabajos para que aquellos que tengan mayor importancia sean procesados en nodos con más y mejores recursos.
- **Aplicaciones**. Se ejecutarán sobre el middleware y se encargarán de procesar el trabajo recibido por éste.

3.2.8.Concurrencia

Durante el estudio de las distintas arquitecturas de red para sistemas distribuidos se ha presentado el concepto de concurrencia en el modelo cliente-servidor, a continuación se define de forma más detalla en qué consiste esta técnica.

La concurrencia acepta dos definiciones, por un lado es la capacidad que tienen los ordenadores de ejecutar distintas aplicaciones de forma simultánea, y por otro lado es la capacidad de interacción y comunicación entre procesos ejecutados al mismo tiempo de acceder al mismo recurso de forma simultánea, siendo esta segunda definición la que interesa de cara al impacto que en el diseño e implementación se habrá de enfrentar.

La concurrencia puede ser a nivel de proceso y a nivel de thread.

Respecto a los procesos, son programas en ejecución formados por las instrucciones que serán ejecutadas para llevar a cabo su función, el estado de ejecución en un momento dado, la memoria reservada para almacenar su contenido e información que servirá para su planificación. Aplicado a la concurrencia, cuando se tienen múltiples procesos en ejecución, se tienen múltiples conjuntos de los elementos comentados.

Respecto a los threads (procesos ligeros), se tratan de un mismo proceso que se ramifica en distintos hilos de ejecución que corren de forma paralela y que comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

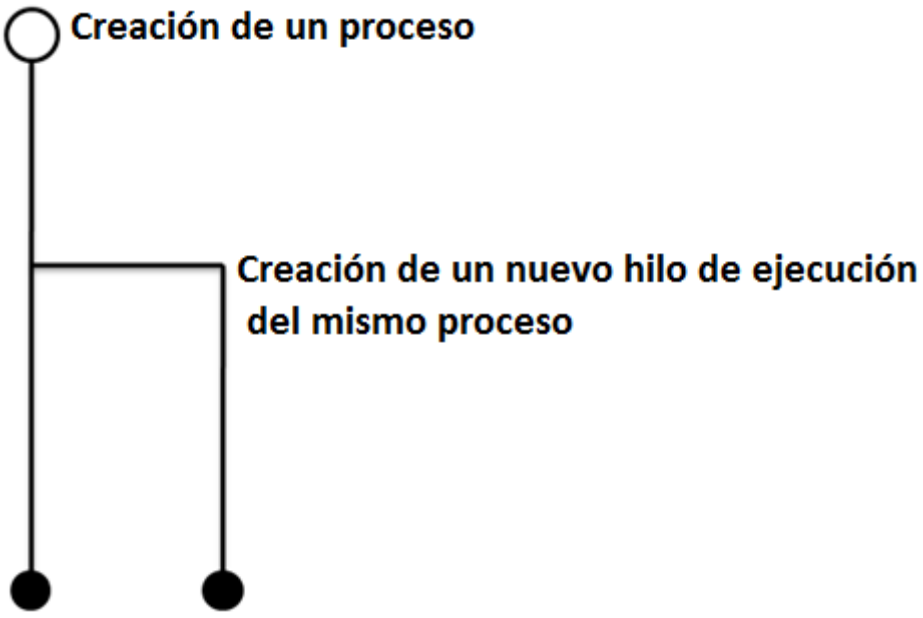

Imagen 16. Esquema de aplicación concurrente

Los problemas derivados de la ejecución no concurrente, se pueden ver con el siguiente ejemplo:

Suponiendo un juego sencillo en el que los jugadores únicamente puedan moverse por el universo presentado por éste, en el lado del cliente se podría tener un bucle como el siguiente:

1. Calcular la posición actual del jugador
2. Enviar su nueva posición
3. Recibir las posiciones de los otros jugadores
4. Pintar la pantalla los datos de los jugadores

El problema que surge es que la aplicación calculará la posición del jugador local, enviará el resultado al servidor y quedará a la espera de que el servidor le responda las coordenadas de los otros jugadores.

Desde el punto de vista de la aplicación servidora, si tiene un único proceso para atender las peticiones de los jugadores, el cliente enviará la coordenada al servidor y quedará bloqueado hasta que sea su turno en servidor y le responda a su petición con las coordenadas del resto de jugadores.

Desde el punto de vista de la aplicación cliente se presenta el mismo problema, se producirá un bloqueo debido al intervalo de tiempo sucedido desde el envío de sus coordenadas hasta que se recibe la respuesta, este tiempo se denomina latencia.

Una solución concurrente a este problema pasa por el uso de procesos ligeros en lugar de tener un único proceso que se encargue de implementar toda la funcionalidad.

Se podría dividir el problema en 3 partes y asignar cada una de estas partes a un hilo de ejecución, la subdivisión propuesta es la siguiente:

- Un hilo encargado del cálculo de la posición del jugador y de pintar los gráficos en pantalla
- Un hilo que envíe las coordenadas al servidor cada X tiempo
- Un último hilo que reciba las coordenadas de los otros jugadores

De este modo se evitará el bloqueo en la representación de los gráficos por el envío y por la recepción de datos.

Por otro lado, esta ejecución concurrente introduce una nueva problemática, ya que al tener varios procesos ligeros de un mismo proceso ejecutándose a la vez será necesario compartir recursos entre ellos.

En el ejemplo presentado anteriormente se tenía un proceso ligero encargado del cálculo de la posición del jugador y de pintar a los jugadores, y otro proceso ligero que enviaba la información de las coordenadas del jugador local, de modo que los dos procesos accederán a al recurso que contiene los datos de las coordenadas del jugador, surgiendo así el problema de la condición de carrera en el que se puede producir una corrupción en el estado de un recurso por su actualización concurrente por parte de varios procesos ligeros.

Es por eso que al tener recursos compartidos entre varios procesos ligeros, antes de acceder a éstos habrá que introducir algún mecanismo de seguridad que asegure la exclusividad en el acceso.

Como mecanismos de seguridad para garantizar el acceso exclusivo a recursos compartidos y evitar las condiciones de carrera, se pueden definir secciones críticas de código que aseguran que el código definido dentro de esa región no será ejecutado por más de un proceso a la vez. Estas secciones siguen la siguiente estructura:

Indicar inicio de sección crítica
Código a ejecutar dentro de la sección crítica
Indicar finalización de sección crítica.

Aplicando el concepto de sección crítica al ejemplo tratado en el apartado, la solución que se propone sería la siguiente:

Hilo de cálculo de posición:

Si se está accediendo a datos del jugador, ESPERAR hasta que se liberen.
INICIO DE SECCIÓN CRÍTICA - Bloquear acceso a datos del jugador
Calcular posición del jugador
FIN DE SECCIÓN CRÍTICA - Desbloquear acceso a datos del jugador
Dibujar estado del jugador

Hilo de envío de posición:

Si se está accediendo a datos del jugador, ESPERAR hasta que se liberen.
INICIO DE SECCIÓN CRÍTICA - Bloquear acceso a datos del jugador
Enviar posición del jugador
FIN DE SECCIÓN CRÍTICA - Desbloquear acceso a datos del jugador

3.3. Librerías de desarrollo de videojuegos

3.3.1. Descripción de una librería de desarrollo de videojuegos

Una librería de desarrollo (también conocida como API) es una interfaz implementada mediante software que permite ser utilizado a través de otro software, con el fin de facilitar a quien las implemente una serie de herramientas para la resolución de las aplicaciones ofreciendo una capa de abstracción.

Son implementadas por aplicaciones, bibliotecas y sistemas operativos para tener acceso a los distintos servicios ofrecidos por el sistema operativo, que puede incluir datos específicos para rutinas, estructuras de datos, funciones, procedimientos, clases de objeto, métodos, protocolos de comunicación, etc.

El objetivo que tienen las librerías de desarrollo de videojuegos es ofrecer una capa de abstracción sobre la que el desarrollador pueda basarse utilizando las herramientas y facilidades proporcionadas por esta (ya sean funciones y procedimientos, clases y métodos, etc.), para enfrentar la problemática introducida en el ámbito del videojuego, dando por ejemplo herramientas para resolver sistemas de colisiones, representar y trabajar con espacios en 2 y 3 dimensiones, introducir efectos gráficos en los modelos presentados en el juego, etc.

3.3.2. Descripción de un motor de juego

Un motor de juego es una serie de rutinas de programación que permiten el diseño, creación y representación de un videojuego. Puede ser presentado a través de un programa de ordenador y/o conjunto de bibliotecas para simplificar y abstraer el desarrollo de juegos u otras aplicaciones con gráficos en tiempo real.

La funcionalidad típica suministrada por un motor de juego incluye: un motor gráfico encargado del renderizado de gráficos 2D y/o 3D, un motor para simular física como por ejemplo la gravedad o la detección de colisiones, soporte para animación de modelos, carga de sonidos, implementación de inteligencia artificial, comunicaciones, gestión de memoria, gestión del sistema de ficheros, soporte de gráficos de escena y entidades, soporte de lenguajes de scripting, etc.

Para ofrecer esta funcionalidad, algunos de los motores de juegos hacen uso de fondo de librerías de desarrollo como Direct3D[16], OpenGL[17], etc.

El objetivo de los motores de juego, además de dar una solución a la problemática habitual del desarrollo de videojuegos, es ofrecer unas herramientas de desarrollo que pueden estar integradas en un IDE; abstraer el uso de los distintos dispositivos hardware como las GPUs, joysticks, teclado, etc; abstraer la plataforma final de ejecución que puede ser un ordenador o una consola a través de minimizar el código a ser modificado.

Estos objetivos variarán en función de la complejidad del motor de juego, ya que mientras que los hay divididos por módulos para que sea el desarrollador quien indique qué se va a utilizar y qué no para generar un motor de juego más personalizado, también los hay que únicamente manejan el renderizado en 3 dimensiones dejando el resto de problemas a la resolución del desarrollador, a estos últimos se los conoce como motores de gráficos.

La principal diferencia de los motores de juego con las librerías de desarrollo es que mientras que las librerías ofrecen una capa más abstracta sobre la que el desarrollador tendrá que resolver los diferentes problemas introducidos por el desarrollo de videojuegos, los motores de juego sí que ofrecen una solución a gran parte de los problemas habituales presentados en los videojuegos además de que por regla general incorporan herramientas para el desarrollo de los mismos (modelado de personajes y escenarios, mecánicas en base al tipo de juego, físicas, edición de sonido, etc).

3.3.3. Resumen de librerías y motores de juego más populares del mercado

A continuación se detallan algunas de las librerías y motores de juego más populares con sus características principales:

- **DirectX**

Descripción:

Es una colección de librerías de desarrollo (API) para la plataforma de Microsoft. Su objetivo es facilitar la inclusión de diversos elementos enfocados especialmente al desarrollo de videojuegos. Actualmente se encuentra en la versión 11.

Algunos de los componentes que incluye y su funcionalidad es la siguiente:

- **DirectDraw**. Actualmente nombrad como Direct2D, se encarga de la representación de gráficos en 2 dimensiones.
- **Direct3D**. Se encarga de la representación de gráficos en 3 dimensiones.
- **DirectWrite**. Se encarga de la representación de las fuentes instaladas en el sistema operativo.
- **DirectCompute**. Se encarga del manejo de la computación que realizará el procesador de la tarjeta gráfica (GPU).
- **DirectInput**. Se encarga de detectar la interacción del usuario con el equipo a través del teclado, ratón, joystick, etc.
- **DirectPlay**. Se encarga de las comunicaciones a través de la red
- **DirectSound**. Se encarga de la reproducción y grabación de sonidos.
- **DirectSound3D**. Se encarga de la reproducción de sonido 3D.

Licencia:

La licencia tanto para desarrollo como para comercialización de productos que hagan uso de DirectX es gratuita.

Requisitos:

Sistemas operativos Microsoft Windows en las versiones Windows 7, Windows Vista y Windows Server 2008, la librería DirectX11 y una tarjeta gráfica compatible con esta tecnología.

Lenguajes de programación:

Se puede desarrollar utilizando DirectX con los lenguajes basados en tecnología .NET.

Plataformas de ejecución:

Los productos desarrollados que hagan uso de DirectX podrán ser ejecutados en sistemas operativos Microsoft Windows.

- **SDL (Simple DirectMedia Layer)**

Descripción:

Se trata de una librería de desarrollo gratuita de bajo nivel que permite la representación de gráficos haciendo uso de framebuffer, reproducción de sonidos y control de dispositivos de entrada con los que interactúa el usuario (teclado, ratón, etc.).

La última versión estable es la 1.2, que permite el desarrollo de videojuegos para una amplia gama de plataformas entre las que se incluyen Linux haciendo uso de framebuffer y Xlib, Windows haciendo uso de DirectX, MacOS, etc.

Con el fin de complementar las funcionalidades de esta librería, se han desarrollado otras librerías adicionales que incluyen las siguientes funcionalidades:

- **SDL Mixer**. Gestiona y reproduce sonidos y música en los juegos, soporta los formatos Wave, MP3 y OGG entre otros.
- **SDL Image**. Provee una mayor capacidad de carga de formato de imagen entre los que se encuentran BMP, JPEG, TIFF, PNG, GIF, etc.
- **SDL Net**. Proporciona la funcionalidad necesaria para el uso de comunicaciones a través de la red.
- **SDL RTF**. Permite la apertura y lectura de ficheros de texto en formato RTF.
- **SDL TTF**. Permite el uso de fuentes TrueType en aplicaciones en SDL.

Características:

Algunas de las características incluidas en la última versión de esta librería son: establecer modos de vídeo de cualquier profundidad con conversión en caso de que el hardware no lo soporte; escritura directa sobre el búfer de macro gráfico; crear superficies con fundido alpha que permita transparencias; control y desactivación de eventos para la detección de dispositivos de entrada; reproducción de sonido de 8 y 16 bits en mono y estéreo; sonido en un hilo de ejecución separado; soporte para programación multi-thread con una librería de creación de hilos simple y facilidad en el uso de mecanismos de sincronización basado en semáforos binarios…

Licencia:

No es necesario pagar ninguna licencia ni para el desarrollo ni para la comercialización de un producto que haga uso de esta librería, el único requerimiento que se hace necesario es que al estar bajo licencia GLGPL (GNU Lesser General Public License) será obligatorio indicar que se ha usado SDL bajo licencia GLGPL.

Requisitos:

Para desarrollar sobre SDL será necesario un compilador de C/C++.

Lenguajes de programación:

De forma nativa, SDL sólo ofrece soporte para C++, pero desarrollos de terceros sobre esta librería permiten la programación en diversos lenguajes de programación como BASIC, C#, Java, Lisp, Pascal, Perl, PHP, Python, Ruby…

Plataformas de ejecución:

Los productos desarrollados sobre SDL podrán ejecutarse desde Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. El código contiene adicionalmente la posibilidad de ejecución sobre AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS, y OS/2, aunque estas plataformas no estén oficialmente soportadas.

- **XNA**

Descripción:

La librería de Microsoft XNA hace referencia a las siglas recursivas XNA's Not Acronymed, XNA no es un acrónimo.

Es una librería de desarrollo que puede llegar a ser considerada un motor de juego de muy bajo nivel dado el nivel de abstracción planteado, ya que incluye el ciclo de ejecución para mover el juego y las herramientas necesarias para que el desarrollador enfrente los problemas habituales en el desarrollo de videojuegos pero sin dar soluciones directas, tendrán que ser desarrolladas en la programación del juego.

Actualmente esta plataforma de desarrollo cuenta con una extensa comunidad que produce juegos, tutoriales y todo tipo de contenido que puede ser visitada a través del siguiente enlace: http://creators.xna.com/es-ES/.

Esta librería cuya última versión es la 3.1 está en constante evolución, y durante su crecimiento ha ido ampliando tanto las funcionalidades proporcionadas como la reproducción de audio y vídeo, como el espectro de mercado al que los juegos producidos usando la librería pueden acceder.

Características:

Algunas características incluidas hasta la versión 3.1 son las siguientes: carga y representación de gráficos en 2 y 3 dimensiones; desarrollo multiplataforma Microsoft minimizando el impacto en los cambios a realizar en la programación; en desarrollos de juegos para Zune, uso del acelerómetro integrado en el dispositivo y detección de la interacción del usuario con la pantalla táctil; en desarrollos para XBOX360, soporte para la representación de Avatares utilizados para representar personajes y jugadores en juegos, facilidad en la creación de partidas para varios jugadores enviando invitaciones a través del sistema XBOX Live; reproducción de vídeos durante la ejecución del juego e incrustados en un espacio 3D; utilización de sonido 3D; representación de modelos 3D basados en esqueletos…

Licencia:

La licencia tanto de desarrollo como de comercialización es gratuita, con la excepción de que si se quiere publicar el videojuego en Xbox Live o Windows Live habrá que pagar una suscripción de carácter anual.

Requisitos:

Para hacer uso de esta librería será necesario el IDE Microsoft Visual Studio en su versión 2005 o superior (dependerá de la versión de XNA que se utilice) y una tarjeta gráfica con soporte para DirectX 9.0c y Shader Model 1.1.

Lenguajes de programación:

Para desarrollar haciendo uso de esta plataforma se programará en el lenguaje C#.

Plataformas de ejecución:

Está enfocado a la plataforma Microsoft permitiendo desarrollo de juegos para PC, la videoconsola Xbox360 y Zune.

- **OGRE 3D**

Descripción:

OGRE 3D es un motor de gráficos 3D de código abierto. Programado en el lenguaje C++, ofrece un motor flexible, orientado a objetos, y multiplataforma permitiendo el desarrollo de videojuegos para plataformas Windows, MacOS X o Linux.

El objetivo de este motor es hacer más sencillo el desarrollo de juegos 3D, explotando al máximo las posibilidades de las tarjetas gráficas ya que brinda soporte para vertex y píxel shaders, como HLSL (DirectX), GLSL (OpenGL) y Cg (DirectX/OpenGL).

El motor está muy extendido y es muy valorado por la comunidad de código abierto, además cuenta con la marca de garantía de que se han desarrollado sobre él muchos juegos, algunos de ellos con carácter comercial como Torchlight y Ankh.

Características:

Actualmente se encuentra en la versión 1.7.1, y algunas características incluidas hasta la fecha son: facilidad en la programación orientada a objetos minimizando el esfuerzo requerido para el renderizado de escenas en 3 dimensiones de forma independiente a la librería de bajo nivel Direct3D/OpenGL; extensa documentación de todas las clases proporcionadas por el motor, soporte para múltiples plataformas (Windows usando C++, Linux usando gcc 3+, Mac OSX usando XCode); soporte para shadders a bajo y alto nivel (ensamblador y Cg, DirectX9 HLSL, etc); soporte para técnicas de múltiples efectos sobre materiales que serán manejadas de forma automática por la que Ogre considere más eficiente para la tarjeta gráfica; actualización en tiempo real de texturas desde formatos PNG, JPEG, TGA, BMP y DDS; soporte para el uso de múltiples formatos de mallas separando los conceptos de buffers de vértices, índices, declaraciones y mapeados; soporte para animación de esqueletos con múltiples huesos de pesos variable, control manual, etc.; gestión configurable de escenas basadas en clases predefinidas o libertad de creación de una clase nueva para obtener control total de la escena; jerarquía de elementos mostrados en la escena para mantener estructuras, movimientos, etc; sistemas de partículas; facilidad para el uso de skyboxes, skyplanes, etc.; manejo automático de transparencias en los objetos…

Licencia:

El desarrollo utilizando este motor es gratuito, y en el caso de que se quiera comercializar el producto desarrollado se sujetará a unas condiciones en función de la versión del motor, siendo para versiones inferiores a 1.7 licencia LGPL y sólo será necesario indicar si ha sido modificado alguna parte del motor para el desarrollo, y en caso de versiones superiores a 1.7 será necesario incluir el texto del MIT del enlace http://ogre.svn.sourceforge.net/viewvc/ogre/trunk/COPYING?revision=9087 en alguna parte del producto (documentación, en la misma aplicación, etc.).

Requisitos:

Para desarrollar haciendo uso de esta librería será necesario una versión de Microsoft Windows XP o superior (Vista o 7), Linux o Mac OS X y en función de la plataforma seleccionada, un compilador de su correspondiente lenguaje.

Lenguajes de programación:

De forma oficial, para plataformas Microsoft se podrá usar C++, en Linux gcc 3+, y en Mac OS X Xcode.

Plataformas de ejecución:

Sistemas operativos Microsoft Windows XP en adelante, Linux y Mac Os X.

- **Game Maker**

Descripción:

Game Maker es un motor de juego que permite el desarrollo rápido de juegos, basado en un lenguaje de programación interpretado y un paquete de desarrollo de software para tal fin.

Este motor de juego que fue creado por el profesor Mark Overmars en el lenguaje de programación Delphi y orientado a usuarios con pocas nociones de programación, permite la creación de videojuego en 2 dimensiones a través de un sistema de fácil uso basado en arrastrar y soltar elementos en la interfaz gráfica ofrecida por la aplicación, incluye además un conjunto de bibliotecas de acciones estándar que cubren movimientos, dibujo básico de sprites, control de estructuras, etc.

De este modo no se exige al programador introducir ninguna línea de código salvo que quiera hacer uso del lenguaje interpretado por este motor, conocido como GML.

Características:

El programa es gratuito, aunque existe una versión comercial ampliada con características adicionales como el soporte nativo para gráficos 3D, uso de partículas y funciones de dibujo avanzadas.

Actualmente se encuentra en su versión 8.0 y algunas de las características que incluye son: capacidad de uso del lenguaje interpretado GML para ampliar las funcionalidades básicas de Game Maker permitiendo llegar a desarrollar juegos en 3D; el usuario puede construir sus propias bibliotecas de acciones para ampliar las ya incluidas en la aplicación; desarrollo simplificado basado en recursos que pueden ser gráficos, sonidos, imágenes de fondos, etc, que serán asignados a los objetos que representarán el mundo del juego; capacidad de detección de eventos como la interacción del usuario con los dispositivos de entrada; programación del juego simplificada basada en arrastrar y soltar elementos que serán los objetos del juego que realizarán acciones; sintaxis de GML muy flexible para facilitar la programación haciendo uso de ella.

Licencia:

La licencia de desarrollo es gratuita aunque existe una versión de pago que amplía las funcionalidades incluidas en la versión libre.

En el caso de la comercialización también se dispone de una licencia gratuita sujeta a determinadas condiciones, como por ejemplo no infringir la propiedad intelectual de otros productos, no desarrollar un videojuego que pueda ser dañino para el equipo donde se ejecuta o incluir de algún modo que el producto fue desarrollado con Game Maker.

Requisitos:

Tener instalada una versión de los sistemas operativos de Microsoft.

Lenguajes de programación:

El desarrollo del videojuego se realizará arrastrando, soltando y configurando los elementos a través de las herramientas propuestas por el editor de Game Maker, aunque adicionalmente se dispone del lenguaje de scripting GML para ampliar las funcionalidades del motor.

Plataformas de ejecución:

Sistemas operativos Microsoft Windows.

- **Torque Game Builder**

Descripción:

Se trata de un motor de juego de pago para el desarrollo de videojuegos en 2 dimensiones que al igual que Game Maker permite la creación de videojuegos a través de las herramientas que propone desde la interfaz gráfica de la aplicación, y que permite añadir funcionalidades a través de un lenguaje de scripting propio que de fondo hará uso del lenguaje de programación C++.

Cuenta también con una versión para el desarrollo de videojuegos en 3 dimensiones llamada Torque3D de funcionamiento similar, basándose en la inclusión de modelos y creación de escenarios cargando los elementos a través de la interfaz de la aplicación.

Características:

Algunas de las características que incluye este motor de juego son: editor de niveles basado en herramientas incluidas en el GUI de la aplicación permitiendo la colocación y manipulación de objetos, efectos de partículas, manejo de tilesets, uso de sonido y vídeo, etc.; uso de técnicas comunes de renderizado de gráficos en 2 dimensiones que se adaptará en función del hardware del usuario final; permite la inclusión de fragmentos de código desde el editor de la aplicación para ampliar las funcionalidades básicas proporcionadas por el motor; inclusión de comunicaciones en red para el desarrollo de videojuegos basados en sistemas de turnos; posibilidad de hacer uso de un lenguaje de scripting con sintaxis similar a C++ para cálculos matemáticos, lectura y escritura de ficheros, manipulación de objetos, etc; incluye cálculo de físicas como inercia, velocidad lineal, fricción, velocidad angular, etc; soporte para efectos de sonido y música en 2 dimensiones permitiendo el uso de múltiples canales, control de volumen, etc; cuenta con diversos kits de inicio para los juegos ofreciendo el comportamiento habitual para juegos de aventura, carreras, plataformas…

Licencia:

La licencia de desarrollo es de pago, y este desembolso variará de si se trata de una licencia de desarrollo indie (dirigida a una única persona trabajando con el motor) o si es una versión de empresa (dirigida a que distintos desarrolladores dentro de una misma empresa compartan la misma licencia.

En el caso de la licencia de comercialización es gratuita ya que se ha pagado la licencia de desarrollo que incluye la puesta en venta del producto en sus dos versiones (indie/empresa).

Requisitos:

Un sistema operativo Microsoft Windows XP en adelante con Microsoft Visual Studio Express 2005/2008 y un compilador de C++, Linux con un compilador de C++ o Mac Os X con un compilador de Xcode.

Lenguajes de programación:

El desarrollo del videojuego se realizará arrastrando, soltando y configurando los elementos a través de las herramientas propuestas por el editor de Torque Game Builder, aunque adicionalmente se dispone de un lenguaje de scripting similar a C++ que amplía las posibilidades nativas del motor.

Plataformas de ejecución:

Permite el desarrollo de videojuegos multiplataforma, pudiendo generarlos para PC, Mac, XBOX 360, Wii, iPhone y tecnologías Web, necesitando como requerimientos en caso del desarrollo sobre plataformas de Microsoft un compilador de C++, el sistema operativo Windows XP o Vista y al menos la versión de Microsoft Visual Studio Express; y para el caso de Mac OS, la versión del sistema operativo 10.3 y XCode.

- **Irrlicht**

Descripción:

Se trata de un motor de juego en 3 dimensiones gratuito programado en C++ que permite el desarrollo de videojuegos en el lenguaje de programación C++ y lenguajes de la plataforma .NET y dada sus características de código abierto permite la programación no oficial en lenguajes como Java, Perl, Ruby, Python, Delphi e incluso Game Maker.

A diferencia de otros motores de juego, no incluye editores que son básicos para el desarrollo de videojuegos como editores de sonido, modelos, niveles, etc, no obstante desde la web en el enlace http://irrlicht.sourceforge.net/toolset.html proponen diferentes herramientas para trabajar estos aspectos.

Características:

Actualmente se encuentra en su versión 1.7.1, y algunas de las características que incluye son: rendimiento mejorado de renderizado 3D en tiempo real haciendo uso de Direct3D y OpenGL; programación independiente de la plataforma; soporte para vertex shader y pixel shader; gestión configurable de escenas basadas en nodos; sistema de animación de personajes basado en esqueletos; efectos de partículas, luces, mapeado del entorno; potente y configurable interfaz gráfica en 2D con botones, listas, cuadros de edición, etc para incluir en el juego; amplio soporte para la carga de formatos de mallas como OBJ, 3DS, B3D, BSP, MD2, X, etc; soporte de texturas en BMP, PNG, PSD, JPG, TGA, etc; fácil uso de sistema de detección de colisiones; inclusión de efectos gráficos más habituales como superficies animadas de agua, luces y sombras dinámicas, transparencia de objetos, luces de mapas, animación de texturas, skyboxes, nieblas, sistema de partículas configurables para nieve, humo, fuego, etc; soporte para múltiples versiones de Direct3D y OpenGL; sistema jerárquico de nodos en la escena para simplificar su manejo permitiendo al desarrollador control total de los elementos de la escena y que permite la carga de nuevos nodos en la escena a partir de mallas, cargadores de texturas, elementos GUI, objetos geométricos, etc…

Licencia:

Tanto el desarrollo como la comercialización de un videojuego desarrollado haciendo uso de este motor es gratuito, con el único añadido de introducir en alguna parte del producto que se ha usado la librería gratuita de JPEG IJG (Independent JPEG Group).

Requisitos:

Una versión del sistema operativo de Microsoft Windows igual o superior a Windows XP, Linux, Mac Os X o cualquier sistema operativo con soporte para la librería de desarrollo SDL que fue comentada anteriormente.

Lenguajes de programación:

Esta librería permite el desarrollo en los lenguajes C++, C#, Java, Perl, Ruby, Basic, Python…

Plataformas de ejecución:

Es multiplataforma, de modo que permitirá generar videojuegos que hagan uso de las librerías Direct3D y OpenGL y que se ejecuten sobre distintas versiones de sistemas operativos de Microsoft Windows (NT, 2000, XP…), Linux y Mac OS.

- **Unreal Engine**

Descripción:

Se trata de un motor de juego para la creación de videojuegos escrito en lenguaje C++ ampliamente conocido en el ámbito del desarrollo de juegos por los títulos en el mercado que se han creado haciendo uso de él, como Unreal Tournament, Gears of War, Bioshock, Borderlands, etc.

Para el desarrollo de videojuegos ofrece una serie de editores como herramientas para la creación del contenido que será usado en los juegos. Estos editores permiten desde la creación de niveles hasta la de personajes y animaciones permitiendo la carga de texturas, pasando por la creación de materiales.

Características:

Algunas de las características que incluye en su última versión son: renderizado en 3 dimensiones en múltiples hilos; soporte para las últimas tecnologías de representación de iluminación por pixel incluyendo mapeo normal, luz direccional, sombras, atenuación de luces, efectos anisotrópicos, etc; análisis de estructuras del entorno en tiempo real que permite la destrucción de estas; soporte para efectos de post-procesado como blur, bloom y depth of field; deformación de materiales a través de vertex shader; animación de modelos basada en esqueletos; herramientas para la navegación y organización entre animaciones y mallas de los modelos; soporte para la reproducción de sonidos y música haciendo uso de posicionamiento 3D; uso de sonidos de ambiente; físicas que permiten la simulación de ropa, efectos ragdoll y simulación de cuerpos para representar objetos deformables y elásticos; soporte para la tecnología de Nvidia PhysX; rutinas de IA; librerías de comunicaciones a través de la red; permite la edición y posterior introducción de cinemáticas durante el juego; para el desarrollo de contenidos dispone de editores de terrenos, materiales, mallas, animaciones, físicas, cinemáticas, scripting para ampliar la capacidad del motor, sonidos… y plug-ins para importar modelos desde otras aplicaciones de modelado 3D como por ejemplo 3D Studio Max, Maya y XSI…

Licencia:

La licencia de desarrollo es gratuita, y en caso de perseguir fin comercial de forma directa o indirecta exige el pago de una licencia comercial en la que se establece que un porcentaje de los ingresos generados por el producto serán para UDK.

Requisitos:

Sistema operativo Microsoft Windows.

Lenguajes de programación:

Para el desarrollo del producto cuenta con herramientas desde las que generar y cargar el contenido, aunque de manera adicional cuenta con un lenguaje de scripting orientado a objetos similar a lenguajes de programación como Java.

Plataformas de ejecución:

En función de la versión utilizada del Unreal Engine se podrá realizar desarrollar para una u otra plataforma, siendo posible desarrollar con Unreal Engine 2 para PC (Windows y Linux), Mac Os, Xbox y PlayStation2; y con Unreal Engine 3 para PC basado en DirectX10, Xbox 360 y PlayStation 3.

4.Manual de desarrollo de infraestructura MMO

En los siguientes apartados se presentará por un lado toda la problemática que introduce la resolución de una infraestructura MMO, desde cómo influye la naturaleza del tipo de juego que se va a desarrollar hasta el protocolo de comunicación a utilizar, pasando por la seguridad en la transmisión de la información o la selección de la forma de almacenamiento de datos; y por otro lado se mostrarán las posibles alternativas de resolución a estos problemas así como la solución más óptima al tipo de juego concreto.

Por ello y dado que a la hora de definir una infraestructura hay que tener en cuenta el sistema final que se va a implementar, en el apartado de solución del problema se supondrá la resolución del sistema para una aplicación final MMO de tipo social, aunque en el análisis del problema se plantearán las diferentes posibilidades que se pueden presentar y se realizará un breve estudio de éstas.

4.1.Infraestructura MMO, análisis del problema

Durante el estudio de un proyecto de implementación de infraestructura para dar soporte a un juego MMO es necesario estudiar distintas problemáticas para que el resultado final sea el adecuado, estos son:

4.1.1.Tipo de juego

El tipo de juego también será determinante a la hora de seleccionar qué arquitectura utilizar y sobre qué protocolo de transporte comunicar la información. Según los estilos de juego, las características deseables serán las siguientes:

- **Shooters**

Los juegos de acción de tipo FPS (First Person Shooter – Disparos en primera persona) consisten en una experiencia que se desarrolla desde la perspectiva del personaje protagonista y en la que el objetivo del jugador es acabar con el resto de jugadores existiendo una gran variedad de modos de juego para conducir este primer objetivo: asalto a fortalezas, captura de banderas, mantenimiento de zonas, todos contra todos, duelos por equipos, etc. Ejemplos de este tipo de juegos en su versión multijugador masiva son Neocron[18], Mag[19] o Huxley[20].

Este tipo de juegos debido a la rapidez con la que se desarrollan, requieren que los jugadores cuenten con la última información sin que sea tan relevante el conocer el estado anterior de un jugador; también puede ser beneficioso para el juego comprobar la información enviada por los jugadores para garantizar que no se tienen estados inconsistentes y que estos no realizan trampas.

Por lo cual las arquitecturas de red que mejor resolverían estos problemas son aquellas que tengan el mínimo de intermediarios y reduzcan el número de comunicaciones, por ejemplo un modelo cliente-servidor; en cuanto al protocolo de transporte, al primar la

velocidad en la transferencia de información y al interesar la información más actual, UDP ofrece el servicio necesario.

- **Conducción**

Los juegos de este tipo suelen presentarse al jugador con una cámara en primera o tercera persona mostrando un vehículo con el cual competir contra otros jugadores pudiendo tender hacia la simulación e introducir al jugador en una experiencia más orientada a los detalles de la conducción como controlar distancias de frenado para tomar curvas correctamente, relación de marchas, estado de los componentes del vehículo, etc; o buscar una experiencia más arcade tendiendo hacia unas físicas menos realistas en busca de una jugabilidad menos rígida y dando lugar a situaciones más fantásticas. Ejemplos de este tipo de juegos en su versión multijugador son: Drift City[21], Trackmania[22] o Darkwind: War of Wheels[23].

En cuanto a la infraestructura a la que tienden, se pueden ver como un caso similar a los de tipo shooter ya que del mismo modo interesa la información más actual de los jugadores no cobrando tanta relevancia los estados anteriores frente a tener la última información de los jugadores.

- **Deportivo**

Los juegos de tipo deportivo presentan al jugador la posibilidad de competir contra otros jugadores en todo tipo de deportes permitiendo la posibilidad de realizar ligas online sobre éstos, partidas rápidas basadas en sistemas de puntuaciones, etc. Ejemplos de este tipo de juegos en su versión multijugador son: Project Powder[24], Free Style Street Basketball[25] o Fantasy Tennis[26].

Respecto a la infraestructura necesaria para dar soporte a este tipo de juego, cabe decir que debido a la limitación en la participación de jugadores por las características que cumplen los deportes, no alcanzaría un nivel masivo de jugadores conectados en la misma partida por lo que no se puede considerar MMO.

Si se planteara el problema de resolver un modo multijugador habría que tener en cuenta que, por el componente estratégico que envuelve a los deportes, es importante tener plena constancia de todos los estados producidos en el juego y habría que mantener un flujo constante del estado del jugador opuesto.

- **Estrategia**

Los juegos de estrategia son aquellos en los que la victoria del jugador se alcanza por el uso de la inteligencia, habilidades técnicas, de planificación y de despliegue. Pueden ser de estrategia en tiempo real, basados en turnos, de si incluyen algún elemento al azar, etc. Algunos ejemplos de este tipo de juegos en su versión multijugador son: Travian[27], Beyond Protocol[28] o Ultracorps[29].

Tanto si se trata de estrategia en tiempo real como basada en turnos, en este tipo de juego será importante saber el estado actual sin perder información de estados anteriores del

jugador; por otro lado, se trata de un género cuya acción tiende a desarrollarse con menor rapidez que en los otros casos expuestos con la excepción de la estrategia en tiempo real que puede precisar una conexión que ofrezca una baja latencia; por último puede ser beneficioso que se comprueben las acciones de los jugadores para evitar estados inconsistentes y el uso de trampas.

A la vista de estas características, la arquitectura de red que mejor se ajusta es nuevamente un modelo cliente-servidor, aunque en el caso de la estrategia por turnos y si la lógica es demasiado compleja se puede optar por un modelo N-Tiered ya que el servidor se puede permitir tiempo de procesamiento entre turnos; respecto al protocolo de transporte, al ser importante recibir todos los estados y decisiones de los jugadores y desarrollarse la partida de forma más pausada, lo ideal sería el uso de TCP.

- **RPG**

Los juegos de tipo RPG (Role Playing Game) se caracterizan por la experiencia que ofrecen al jugador basando el juego en un sistema de experiencia en el que a través de objetivos se avancen niveles y se permita evolucionar al personaje a decisión del jugador. Las temáticas de este tipo de juegos tocan prácticamente todas las áreas, además de que son conjugables con otros tipos de juegos, permitiendo la combinación por ejemplo con juegos de estrategia dando la posibilidad al jugador de conforme avanza la experiencia jugable desarrollar su personaje/s en un determinado ámbito, o en juegos de acción tipo shooter permitiendo desbloquear nuevas armas, etc. Ejemplos de este tipo de juego son: World of Warcraft[30], City of Heroes[31] o Ragnarok[32].

Este tipo de juegos se puede ver como un caso particular del anterior ya que del mismo modo toda información de los jugadores es relevante y no podemos perder estados actuales ni anteriores de los jugadores; la acción se desarrolla a un ritmo pausado; y es conveniente confirmar la información recibida para evitar el uso de trampas.

De estas características se deduce lo mismo que en el caso anterior, la arquitectura de red que mejor se ajusta es un modelo cliente-servidor, y el protocolo de transporte idóneo es TCP.

- **Social**

Los tipos de juego social pueden no considerarse tanto un juego como un punto de encuentro virtual para los participantes, aunque el objetivo final sea entretener a los participantes sustituyendo la jugabilidad por la posibilidad de interacción entre la población. Ejemplos de este tipo de juegos son: Second Life[33] o PlayStation Home[34].

Este tipo si bien por su objetivo no se parece a ninguno de los descritos anteriormente, la solución desde el punto de vista de la implementación es muy similar ya que del mismo modo toda información de los jugadores es relevante y no podemos perder estados actuales ni anteriores de los jugadores ya que lo que prima en este tipo son las acciones de los participantes; la acción se desarrolla a un ritmo pausado; y es conveniente confirmar la información recibida para evitar el abuso de normas establecidas.

De estas características se deduce lo mismo que en el caso anterior, la arquitectura de red que mejor se ajusta es un modelo cliente-servidor, y el protocolo de transporte idóneo es TCP.

- **Conclusión**

De este análisis se puede ver que en función de la velocidad a la que se va a desarrollar la acción de la "partida" y la necesidad de garantía en la recepción de la información se optará por una infraestructura de comunicaciones u otra.

4.1.2.Almacenamiento de datos, ficheros frente a bases de datos

Dado que en este tipo de aplicaciones la información del jugador se guarda en el servidor por motivos de seguridad, es importante plantear el problema de almacenamiento, el cual puede ser resuelto mediante el uso de ficheros o de bases de datos.

Según el estudio aplicado al desarrollo de MMORPGs por Radu Privantu[35], las ventajas e inconvenientes que estas alternativas presentan son:

- **Ficheros**

Las ventajas que aportan los ficheros son: velocidad en lectura y escritura sobre ficheros de pequeño tamaño, facilidad de implementación, no es necesario el uso de librerías específicas.

Los inconvenientes que presenta son: costosa actualización si se quieren incluir o modificar campos en la estructura de los ficheros, pérdida de eficiencia en procesos de actualización, procesos de copia de seguridad y restauración más complejos.

- **Bases de datos**

Las ventajas que aportan las bases de datos son: facilidad para incluir y modificar campos, mayor facilidad en la actualización de estadísticas de jugadores, con el uso de sentencias SQL se puede recuperar información de distintas tablas de forma eficiente, facilidad para la copia de seguridad y restauración.

Por el contrario, los inconvenientes que presenta son: dependencia con una base de datos (actualizaciones, seguridad, etc.), facilidad para cometer errores en sentencias SQL, introducción de retardos en la recuperación de información al acceder a una base de datos, posible necesidad de introducir código adicional para convertir la información a la entrada y a la salida de la base de datos, necesidad de una librería específica para acceder a la base de datos, la corrupción del fichero/s que contiene la base de datos puede suponer la pérdida de toda la información almacenada.

- **Conclusión**

Presentadas estas ventajas e inconvenientes donde salvo pequeñas excepciones las ventajas de una opción son los puntos débiles de la otra, se ha de tener en cuenta también otras características como son el tipo de juego y si este va a necesitar la recuperación de múltiples datos durante la partida (juegos de tipo rpg) o si tendrá una primera carga más grande (juegos de tipo acción), generación de estadísticas, etc.; la cantidad y estructura de la información almacenada por el jugador en el servidor; la infraestructura económica de la que se dispone para la adquisición de licencias de sistemas gestores de bases de datos y su instalación en un servidor adecuado; y el número estimado de jugadores que se prevé que harán uso de la plataforma.

De acuerdo a estas características en este manual se optará por la implementación del almacenamiento a través de ficheros dado que se dará soporte a un número reducido de

jugadores, durante la partida la única carga de datos se realizará al inicio y no se tiene una estructura compleja en la información almacenada respecto a cada jugador.

4.1.3.Arquitectura de red

Ya que un juego o aplicación MMO tendrá que dar soporte masivo a jugadores a través de la red, el sistema que dará solución al problema es deseable que sea un sistema distribuido por cumplir las características que estos presentan:

- **Escalabilidad**. Porque a medida que los clientes crezcan es necesario que crezcan los servidores para soportar y balancear la carga de los jugadores.

- **Transparencia.** Porque el jugador no tiene porque saber a qué servidor está conectado o como se está procesando su información, sólo necesita saber su información y la del resto de jugadores

- **Tolerancia a fallos.** A nivel de hardware porque si un servidor deja de responder lo ideal es que otro le sustituya.

- **Concurrencia.** Porque habrá múltiples jugadores conectados a los servidores.

- **Recursos compartidos.** Desde el punto de vista de la información, todos los jugadores tendrán que saber dónde y en qué estado se encuentran el resto de jugadores.

Respecto a las arquitecturas expuestas para dar solución a un sistema distribuido, las ventajas y desventajas que presentan si el objetivo es una infraestructura MMO son las siguientes:

- **Arquitectura par a par (P2P)**

Del lado de las ventajas se tiene la necesidad de escalabilidad, dado que el número de clientes será elevado y esta arquitectura incrementa su eficiencia con el número de nodos participantes; otra ventaja es el hecho de carecer de un nodo principal, ya que desde el punto de vista de la tolerancia a fallos si un nodo dejara de existir en la red podría ser sustituido por otro nodo.

Del lado de las desventajas tenemos la localización, ya que los usuarios interactuarán además de con el entorno (en cuyo caso tendrán que informar al resto de jugadores) con otros jugadores, teniendo que localizar a estos en la red; asimetría en el ancho de banda de los distintos clientes que participen en el juego; y también muy importante es el problema de la seguridad, ya que la información que navega hasta un determinado nodo podrá ser observada y modificada por otros jugadores no garantizando la integridad de los datos salvo que se inviertan recursos en establecer un protocolo de seguridad a la hora de comunicar esta información.

De este análisis se puede concluir que las ventajas presentadas por esta arquitectura no consiguen atenuar el impacto de las desventajas.

- **Arquitectura clúster**

Las ventajas que esta arquitectura ofrece respecto al problema de la infraestructura MMO son la alta capacidad de escalabilidad que puede ofrecer dando así la posibilidad de crecimiento de servidores en base al crecimiento de los clientes; la detección y recuperación ante errores en los servidores de modo que si uno de estos falla se seguiría funcionando mientras quedasen nodos disponibles en la red.

En cuanto a las desventajas se tiene que la administración y mantenimiento de múltiples máquinas es más complejo que si se tratara de una sola máquina con mayor potencia; tampoco se cuenta con sistemas operativos distribuidos con la suficiente estabilidad como para abarcar los conceptos de seguridad y escalabilidad en los sistemas de clustering; y aunque como ventaja se dejaba notar la facilidad para escalar, esta sólo sería aplicable a nivel de comunicación, no dando tan buena respuesta en aplicaciones transaccionales como bases de datos, las cuales son un elemento básico en infraestructuras MMO de gran tamaño.

- **Arquitectura Cliente – Servidor N-Tiered**

Ofrece una solución viable al problema, pero en función de las capas implementadas obtendremos una mejor respuesta, siendo estos casos:

  - **Three-Tiered**

    Separar la lógica de la aplicación, de los datos y de la interfaz de usuario puede parecer aportar un incremento en el rendimiento a primera vista, pero esto es discutible.

    Del lado de las ventajas se tiene que si la lógica del juego es demasiado compleja, el cliente no tendrá que procesarla y quedará en manos de un servidor.

    Del lado de las desventajas está el incremento en el tráfico de red, ya que se tendrá que comunicar también información para procesar la lógica al servidor y recibir su respuesta; y la necesidad de invertir en el lado de los servidores en un hardware que soporte estos cálculos de forma masiva para todos los jugadores.

    De este análisis se puede concluir que esta solución reportará beneficios en caso de que el juego tenga una lógica de gran complejidad y además se cuente con una buena conexión tanto en el lado del servidor como de los clientes.

  - **N-Tiered**

    Las ventajas que presenta esta solución son las mismas que en el caso anterior, podemos dividir el proceso de la información en unidades más pequeñas y que cada servidor se encargue de procesar su información correspondiente.

    Del lado de las desventajas se tiene el problema que resulta de subdividir en más capas, el incremento en la necesidad de comunicar información entre clientes y servidores, resultando en un protocolo que genera un mayor tráfico de red y por

tanto obligando a clientes y servidores a contar con una mejor conexión; además de la complejidad introducida en el diseño del sistema final.

De este análisis se puede concluir que esta solución introducirá más complejidad en el diseño del sistema y tráfico de red de forma innecesaria, dado que a la hora de procesar la información de un juego no se tendrán más frentes que la lógica del juego, los datos, y la interfaz del usuario.

- **Two-Tiered**

Del lado de las ventajas se tiene la centralización del control en uno o varios servidores, lo que facilita la orquestación en las comunicaciones y la seguridad en la información que se transmite ya que esta se puede validar al llegar al servidor antes de reenviarse a otros clientes; la independencia en la escalabilidad, permitiendo el crecimiento de los servidores en base a la demanda de los clientes; y las facilidades de mantenimiento pudiendo desviar las comunicaciones a otro servidor cuando uno esté siendo reparado o modificado.

Del lado de las desventajas se tiene la congestión en el tráfico de comunicaciones que se puede amortiguar con el crecimiento de los servidores observando las demandas de los clientes; la robustez en caso de fallo de un servidor, que aunque es el principal defecto de esta solución, se puede controlar desde el lado del cliente identificando cuando el servidor no responde y conectándose a otro si lo hay; y la necesidad de un hardware específico, cuyo mayor problema viene de soportar a múltiples jugadores y que se podría solucionar igual que en el caso de la congestión, introduciendo más servidores en función del número de clientes.

- **Conclusión**

De este análisis se puede ver que el modelo cliente-servidor (two-tiered) presenta la solución más equilibrada al problema presentado por la infraestructura MMO buscada, ya que no se cuenta con una lógica que exija de una mayor granularidad del problema a resolver, facilitará la localización del resto de jugadores y el control de la información mejorando la seguridad del sistema.

4.1.4.Protocolo de transporte, TCP frente a UDP

Los protocolos TCP y UDP pueden ofrecer una solución al problema planteado por la infraestructura MMO pero la elección final dependerá de cómo se diseñe el sistema final, pudiendo utilizarse para las comunicaciones uno de ellos de forma exclusiva, o una solución híbrida en la que se usen ambos protocolos.

Las ventajas y desventajas que estas posibilidades aportan son las siguientes:

- **Uso exclusivo de TCP**

La principal ventaja que aporta este protocolo es el servicio confiable de entrega de datos, de modo que se tiene la seguridad de que las acciones de los jugadores serán recibidas en el servidor y por extensión en el resto de clientes. Adicionalmente, el control de flujo ofrecido por el protocolo TCP asegurará que el receptor no se desborde por congestión en caso de que el emisor envíe mucha información en intervalos de tiempo cortos.

La desventaja que presenta este protocolo es que al ofrecer un servicio orientado a conexión, si se utiliza una arquitectura de red que no esté basada en un modelo cliente-servidor como por ejemplo una arquitectura P2P, las conexiones y desconexiones que se realicen con el resto de nodos supondrán un incremento en el tráfico de datos a través de la red.

- **Uso exclusivo de UDP**

La ventaja que aporta este protocolo nace de la desventaja vista en TCP, al no ser un servicio orientado a conexión ahorrará tiempo de conexión y desconexión con otros elementos de la red (ya sean servidores, clientes o nodos) y tráfico de red.

Del mismo modo, la desventaja que presenta este protocolo es el opuesto a la ventaja vista en TCP, no garantiza que la información llegue al destinatario permitiendo que acciones que realizan los jugadores no sean informadas al resto de elementos de la red, y no tiene control de flujo por lo que emisores rápidos pueden desbordar a receptores lentos.

El motivo por el que se descarta esta opción es que en este tipo de juegos no es admisible la pérdida de determinada información, como por ejemplo la interacción entre jugadores, el envío de mensajes de chat, etc.

- **Híbrido TCP/UDP**

El uso de ambos protocolos puede servir para encontrar un equilibrio entre las ventajas y desventajas de TCP y UDP como uso exclusivo, permitiendo por ejemplo que la información menos relevante o la que se pueda reconstruir sea enviada a través de UDP (información como coordenadas de posición de los jugadores), y que acciones relevantes de las que se necesite garantía de su llegada sean transmitidas mediante TCP.

El principal problema de usar una solución híbrida entre TCP y UDP está en que ambos protocolos de la capa de transporte corren sobre la capa de red IP, por lo que paquetes

enviados y recibidos desde un protocolo afectarán al envío y recepción del otro protocolo induciendo así a UDP a una mayor pérdida de paquetes, obligando a introducir una lógica más compleja en la aplicación cliente para reconstruir la información perdida.

- **Conclusión**

De este análisis se puede ver que el protocolo de nivel de transporte más adecuado para este tipo de MMO será TCP, ya que asegura que la información será recibida en el servidor y por tanto en el resto de jugadores, y además si se reduce el número de paquetes enviados y si se alcanza un buen equilibrio entre servidores/clientes tampoco se producirá una congestión de la red.

4.1.5.Concurrencia frente a no concurrencia

Cuando se plantea la posibilidad de múltiples comunicaciones de forma simultánea es necesario estudiar cómo va a manejar el servidor esas conexiones no siendo viable el uso de sockets bloqueantes en un único proceso en ejecución ya que la espera de la recepción de datos de un jugador bloquearía la recepción de datos del siguiente.

Las opciones más adecuadas para este tipo de infraestructuras son el uso de sockets no bloqueantes en un único proceso, o el uso de sockets bloqueantes en múltiples hilos.

Las ventajas e inconvenientes que estas opciones presentan son:

- **Sockets no bloqueantes sobre un único proceso**

Algunas ventajas de esta opción son una mayor facilidad de implementación y depuración evitando el uso de mecanismos de sincronización, y un menor tiempo de respuesta en la mayoría de las situaciones.

Las desventajas son que si se cuenta con un elevado número de jugadores conectados y la petición de uno de los jugadores exige de un mayor tiempo de proceso el resto de jugadores quedarán a la espera de que esta finalice.

- **Sockets bloqueantes en múltiples hilos de ejecución**

Las ventajas que esta opción presenta son: mejor aprovechamiento de los recursos, por ejemplo el uso de múltiples procesadores; que a diferencia de la desventaja citada en el caso de un único proceso, no bloquearía necesariamente al resto de jugadores; mayor facilidad en la codificación de los sockets.

El principal inconveniente que presenta es que introduce mayor dificultad en la implementación ya que hay que tener en cuenta mecanismos de sincronización en el acceso a variables compartidas.

- **Conclusión**

De este análisis se concluye que la solución más eficiente para el desarrollo de una infraestructura MMO es el uso de sockets bloqueantes en múltiples hilos en ejecución, ya que si se va a dar soporte a cientos o miles de usuarios a la vez no se puede permitir el bloqueo del servicio debido a uno de los jugadores.

4.2.Infraestructura MMO, solución del problema

4.2.1.Almacenamiento de datos

Según se explicó en 4.1.2. Almacenamiento de datos, ficheros frente a bases de datos, debido a que el juego que se implementará como caso de prueba es de tipo MMO social, y dado que inicialmente no se contará con una gran cantidad de jugadores, se optará por la implementación basada en ficheros ya que las ventajas que aportan las bases de datos en este tipo de juegos no superan a los problemas que éstas introducirían.

Como solución se propone que se generen dos ficheros, uno que contendrá la relación de los jugadores y los datos de sus personajes, y otro con la relación de los jugadores y sus contraseñas de inicio de sesión, de este modo se tendrá una estructura claramente diferenciada que facilitaría una futura migración de los datos del sistema de ficheros a un sistema de bases de datos.

El formato de cada fichero sería:

- **DBJugadores.dat**

```
Usuario1:Personaje1#Personaje2#Personaje3#
Usuario2:Personaje1#Perosnaje2#Personaje3#
…
```

Tabla 1. Fichero base de datos de jugadores

El contenido de cada uno de estos campos será:

- **UsuarioX**
  Nombre de inicio de sesión del usuario

- **PersonajeX**
  Datos de uno de los personajes del usuario, contendrá la siguiente información: Nombre del personaje del jugador, imagen utilizada como textura para ese personaje y coordenadas X e Y donde se encontrará ese personaje al inicio de la siguiente sesión.

- **passwd.dat**

```
Usuario1:Clave
Usuario2:Clave
…
```

Tabla 2. Fichero de contraseñas de usuarios

El contenido de cada uno de estos campos será:

- **UsuarioX**
  Nombre de inicio de sesión del usuario

- **Clave**
  Contraseña de inicio de sesión del usuario

4.2.2.Arquitectura de red, modelo cliente-servidor

Seleccionado como objetivo de desarrollo una aplicación MMO de tipo social, y como se estudió en los puntos 4.1.1. Tipo de juego y 4.1.5. Concurrencia frente a no concurrencia, la arquitectura que mejor se ajustará a esta aplicación final será un modelo cliente-servidor en el que los servidores que ofrezcan una funcionalidad de mayor tiempo de proceso a petición del jugador tendrán ejecución concurrente con sockets no bloqueantes sobre múltiples hilos.

En cuanto a los servidores que darán respuesta a este modelo se dividirán de acuerdo a su funcionalidad y atendiendo a los factores deseados de escalabilidad, transparencia y tolerancia a fallos.

Con este fin se proponen 3 servidores como resultado: un servidor encargado de organizar todas las conexiones y que conozca en todo momento qué otros servidores y clientes están activos para poder balancear la carga y para que los clientes no tengan que conocer esta información; un servidor encargado de manejar los procesos de logueo de los jugadores, que acceda al sistema de almacenamiento para recuperar la información de éstos y que permita autenticar al jugador, seleccionar, crear y eliminar nuevos personajes; y por último un servidor de juego encargado de gestionar la información de los jugadores que se encuentran jugando actualmente y que por tanto comuniquen la información de la ubicación y acción realizadas por un jugador, al resto de jugadores.

En cuanto al cliente se propone la aplicación con la interfaz gráfica que ejecutará el jugador y que se comunicará con los distintos servidores conociendo únicamente y de forma inicial el servidor encargado de organizar las conexiones.

La funcionalidad detallada de estos servidores y cliente, y un pseudocódigo de qué información y cómo se procesará se muestra a continuación:

- **Servidor 1 - Servidor de conexión SC**

El servidor de conexión es el encargado de controlar las conexiones de los clientes y del resto de servidores del sistema distribuido. Las funciones que realizará son:

- Registrar la conexión de un servidor, identificando de qué tipo es, por qué puerto atenderá a los clientes y guardando la dirección IP dónde se encuentra.
- Registra la desconexión de un servidor.
- Registrar la conexión de un cliente e indicarle las direcciones IP del resto de servidores, o informarle en caso de que estos servidores no existan.
- Controlar que los servidores permanecen activos.

Para ello cuenta con un proceso que pondrá un socket TCP a la escucha por un puerto que leerá de un fichero de configuración, estableciendo una cola de conexiones pendientes de hasta un máximo de 10 clientes/servidores.

Dado que este servidor no tendrá una gran carga computacional no es necesario que se ejecute en múltiples hilos, por lo que en el mismo proceso realizará de forma iterativa el siguiente ciclo de ejecución:

PSEUDOCÓDIGO DE SERVIDOR DE CONEXIÓN

```
Creación del socket para binding
El socket abre el puerto indicado con una cola de espera de 10 clientes
Mientras que no se detenga el servicio del Servidor de Conexión
        Se crea un socket para clientes que queda a la escucha
        Se conecta un cliente
        Se recibe un comando del cliente
                Si el comando es de un Servidor de Login
                        Se negocia la conexión/desconexión del Servidor de Login
                Si el comando es de un Servidor de Juego
                        Se negocia la conexión/desconexión del Servidor de Juego
                Si el comando es de un Jugador
                        Se negocia la conexión del Jugador
        Se cierra la conexión con el cliente actual
        Se vuelve al principio del bucle
Se cierra el socket para binding
```

- **Servidor 2 - Servidor de login SL**

El servidor de login es el encargado de autenticar a los jugadores y dirigir el proceso de selección, borrado y creación de personajes. Las funciones que realizará son:

- Autenticar al jugador a través de los datos de nombre de usuario y contraseña, garantizando así que el usuario está dado de alta en el sistema y permitiendo guardar datos de los personajes con los que juega y asociarlos a su cuenta.
- Indicar al jugador que personajes tiene para que seleccione con cuál quiere iniciar la partida, dando la posibilidad de que un jugador tenga varios personajes.
- Permite el borrado y creación de nuevos personajes en el sistema.
- Una vez que el jugador ha seleccionado un personaje el Servidor de Login se lo comunica al Servidor de Juego para que sepa las coordenadas y características del personaje seleccionado, de este modo se evita que el jugador sea quien mande las características del personaje al Servidor de Juego minimizando la posibilidad de realizar trampas.

Puesto que los jugadores se conectarán a este servidor y permanecerán en él hasta que seleccionen un personaje con el que jugar, el servidor contará con:

- Un hilo que pondrá un socket TCP a la escucha por un puerto que leerá de un fichero de configuración, estableciendo una cola de conexiones pendientes de hasta un máximo de 10 clientes/servidores para recibir comandos del Servidor de Conexión.
- Un hilo por cada usuario que pondrá un socket TCP a la escucha por un puerto que generará de forma incremental a partir del puerto de escucha para el Servidor de Conexión.

El pseudocódigo al que atiende este servidor es el siguiente:

PSEUDOCÓDIGO DE SERVIDOR DE LOGIN

-HILO PRINCIPAL-

```
Se crea un socket y se conecta al Servidor de Conexión
        Se envía el puerto por el que el Servidor de Login atenderá
Se cierra el socket con el Servidor de Conexión

Creación del socket para binding
El socket abre el puerto indicado con una cola de espera de 10 clientes
Mientras que no se detenga el servicio del Servidor de Login
        Se crea un socket y queda a la escucha
        Se conecta el Servidor de Conexión
        Se recibe un comando
                Si el comando es de conexión de un nuevo Jugador
                        Se crea un nuevo hilo para atender al jugador
```

Si el comando es de conexión de un Servidor de Juego
Se añade el Servidor de Juego a la lista
Se cierra la conexión con el cliente actual
Se vuelve al principio del bucle
Se cierra el socket para binding

-HILO DE ATENCIÓN DE JUGADOR-

Creación del socket para binding
El socket abre el puerto indicado con una cola de espera de 1 clientes
Se crea un socket para el jugador y queda a la escucha
Se conecta el Jugador
Se recibe la autenticación (usuario + contraseña) del jugador
Mientras la autenticación sea incorrecta
Se solicita una nueva autenticación
Se le envía sus personajes
Mientras se reciba un comando distinto de selección de personaje
Se recibe un comando
Si el comando es de selección de un personaje
Se envía el personaje al Servidor de Juego
Si el comando es de borrado de un personaje
Se borra el personaje
Si el comando es de creación de personaje
Se crea el personaje en la BD si no ha alcanzado el límite
Se cierra la conexión con el cliente actual
Se cierra la conexión para binding

- **Servidor 3 - Servidor de juego SPJ**

El Servidor de Juego es el encargado de recibir el estado de los jugadores y comunicárselo al resto ofreciendo la siguiente funcionalidad:

- Recibe el estado de los jugadores conectados a dicho servidor.
- Comprueba que el estado recibido por los jugadores es posible para evitar el uso de trampas.
- Reenvía el estado recibido de los jugadores al resto de jugadores que pueda interesar esta información, de este modo se controla que jugadores distantes no reciban información que no les afectará y se ahorra comunicaciones.

Puesto que los jugadores se conectarán a este servidor y permanecerán en él hasta que finalicen la partida, el servidor contará con:

- Un hilo que pondrá un socket TCP a la escucha por un puerto que leerá de un fichero de configuración, estableciendo una cola de conexiones pendientes de hasta un máximo de 10 clientes/servidores para recibir comandos del Servidor de Conexión y Servidor de Login.
- Un hilo por cada usuario que pondrá: un socket TCP a la escucha por un puerto que generará de forma incremental a partir del puerto de escucha para el Servidor de Conexión y Login por el que recibirá el estado del jugador; y un segundo socket TCP conectado con el ordenador del jugador para enviarle el estado del resto de jugadores.

El motivo por el que con un mismo usuario se establecen dos conexiones (una en dirección Jugador-Servidor y otra en dirección Servidor-Jugador) es para que estas conexiones no sean bloqueantes, de modo que de forma independiente se puede enviar y recibir información al cliente.

La ventaja de mantener el flujo de información separado garantiza que el jugador no se quedará bloqueado a la espera de que el servidor le responda, ahorrando tráfico de datos ya que sólo se enviará información cuando se produzcan cambios y no como “palabra de paso” para recibir el siguiente estado del jugador.

El pseudocódigo al que atiende este servidor es el siguiente:

PSEUDOCÓDIGO DE SERVIDOR DE JUEGO

-HILO PRINCIPAL-

```
Se crea un socket y se conecta al Servidor de Conexión
        Se envía el puerto por el que el Servidor de Juego atenderá
Se cierra el socket con el Servidor de Conexión

Creación del socket para binding
```

El socket abre el puerto indicado con una cola de espera de 10 clientes
Mientras que no se detenga el servicio del Servidor de Login
    Se crea un socket y queda a la escucha
    Se conecta el Servidor de Conexión
    Se recibe un comando
        Si el comando es de conexión de un nuevo Jugador
            Se crea un nuevo hilo para atender al jugador
        Si el comando es de información del Servidor de Login
            Se anexa la información del jugador a la lista de jugadores conectados
    Se cierra la conexión con el cliente actual
    Se vuelve al principio del bucle
Se cierra el socket para binding

-HILO DE ATENCIÓN DE JUGADOR-

Creación del socket para binding
El socket abre el puerto indicado con una cola de espera de 1 clientes
Se crea un socket para recibir el estado del jugador y queda a la escucha
Se conecta el Jugador
Se cierra la conexión para binding
Se le asigna al jugador las características que ha informado el Servidor de Login
Se conecta un socket al ordenador del jugador para enviarle los datos de los otros jugadores
Mientras haya jugadores en la lista de jugadores conectados
    Se le envía la información del jugador
Mientras se reciban comandos del jugador Y no sea un comando de desconexión
    Se comprueba el comando
    Mientras haya jugadores en la lista de jugadores conectados
        Se reenvía el comando al jugador
Se cierra la conexión con el cliente actual

- **Cliente**

El cliente es la parte de la aplicación que ejecutará el jugador y que se encargará de representar el mundo virtual de una forma gráfica. Esta parte se encargará de las siguientes funciones:

- Conectar con el Servidor de Conexión para verificar que están disponibles los servidores necesarios para iniciar la partida
- Registrar en el Servidor de Login al jugador a través de una interfaz gráfica
- Representar el mundo virtual de forma gráfica
- Enviar los datos con los movimientos y acciones realizados por el jugador
- Representar de forma gráfica los movimientos y acciones realizados por el resto de jugadores
- Calcular lógica de colisiones entre objetos y usuarios

Dado que la comunicación entre jugador y Servidor de Juego requiere dos sockets que no bloqueen el proceso de puesta en pantalla de los gráficos, la aplicación cliente contará con los siguientes hilos:

- Un hilo principal que se encargará de: primero, conectar un Socket con el Servidor de Conexión en el inicio de la aplicación para verificar el estado de los servidores; segundo, gestionar toda la comunicación con el Servidor de Login para el proceso de autenticación; tercero, de crear los dos hilos encargados de enviar y recibir información de los jugadores; y por último, se encargará de pintar en pantalla los gráficos recibidos a través del socket de recepción establecido con el Servidor de Juego.
- Un hilo con un socket de envío conectado al Servidor de Juego que se encargará de enviar las acciones llevadas a cabo por el jugador.
- Un hilo con un socket de recepción conectado al Servidor de Juego que se encargará de recibir las acciones llevadas a cabo por el resto de jugadores

El pseudocódigo al que atiende el cliente es el siguiente:

PSEUDOCÓDIGO DEL CLIENTE

-HILO PRINCIPAL-

```
Se crea un socket y se conecta al Servidor de Conexión
        Se obtienen los datos del resto de servidores
Se cierra el socket con el Servidor de Conexión

Si faltan alguno de los servidores
        Se le indica al usuario que no se puede iniciar la partida y el motivo

Se crea un socket y se conecta al Servidor de Login
Mientras que no se envíe un comando de selección de personaje
```

```
        Se envía el comando del jugador al Servidor de Login
Se cierra el socket

Se crea un nuevo hilo para el envío de datos al Servidor de Juego
Se crea un nuevo hilo para la recepción de datos del Servidor de Juego
Mientras no se termine la partida
        Se obtiene la interacción del jugador local
        Se pone la interacción local en la cola de envíos
        Se recogen los datos de la cola de datos recibidos
        Se pinta el estado actual
Se cierra el socket de envío y recepción

        -HILO DE ENVÍO DE DATOS -

Se crea un socket y se conecta al Servidor de juego
Mientras no se termina la partida
        Se envían los datos puestos en la cola de envíos por el hilo principal
        Se elimina el dato enviado de la cola

        -HILO DE RECEPCIÓN DE DATOS -

Se crea un socket para recibir la conexión del Servidor de juego y queda a la escucha
Se conecta el Servidor de juego
Se cierra la conexión para binding
Mientras no se termina la partida
        Se reciben datos del Servidor de Juego
        Se escriben en la cola de datos recibidos
```

Para no tener un flujo constante de movimientos y ahorrar en comunicación, únicamente se van a mandar los cambios de estado en los movimientos del jugador, así por ejemplo cuando el usuario presione arriba, este movimiento sólo se envía una vez, y no se volverá a mandar nada hasta que no suelte la tecla o presione otra.

4.2.3.Arquitectura de red, seguridad

De forma independiente a si la información comunicada entre cliente y servidores está cifrada o no, es necesario que sea validada al llegar del cliente al servidor ya que de otro modo podría haber sido alterada durante su transmisión o por parte de los jugadores y provocar estados incoherentes o el fallo tanto del cliente como del propio servidor.

Esta funcionalidad recae sobre el servidor con el que el cliente se está comunicando, de modo que cualquier acción que el jugador realice, antes de ser comunicada al resto de jugadores o incluso al mismo jugador, ha de ser validada por el servidor. Las comprobaciones básicas que han de ser controladas son:

- **Tamaño de los paquetes recibidos** por parte del cliente antes de su procesamiento y difusión al resto de jugadores, ya que en caso de que llegue mayor o menor información podrían producir un fallo en el servidor.
- Comprobar la **coherencia de los datos incluidos en el paquete** recibido por parte del cliente, controlando así a bajo nivel que un paquete contiene toda la información que tiene que contener; y a alto nivel, que por ejemplo un jugador que se encuentra en una parte del mapa no realice un salto al otro extremo de éste.

Por otro lado esta solución en la que toda acción pasa por el servidor y se valida antes de que el resto de jugadores incluye otra serie de beneficios como son:

- Si se desea, en el sistema de chat se puede introducir un diccionario de palabras prohibidas, de modo que si un jugador escribe alguna de estas palabras puede ser eliminada antes de llegar al resto de jugadores.
- En caso de que un jugador esté presentando una latencia alta el servidor puede calcular sus coordenadas para corregir los desvíos que se estén produciendo.
- Como se introducía en las comprobaciones básicas, se puede controlar que un jugador no haga trampas, como una técnica que suele ser habitual en MMOs denominada *speed-hack,* en la que el jugador envía al servidor sus coordenadas de posición avanzando a una velocidad superior a la establecida por el juego y permitiéndole ir más rápido que el resto de jugadores; o en juegos de tipo MMORPG en los que de no existir este control un jugador podría utilizar un ítem que realmente no posee, comerciar con un jugador que no se encuentra cerca suya, o recoger un objeto del suelo que realmente no está ahí.

Una vez controlado el aspecto de los datos enviados entre cliente y servidores, es importante estudiar el uso de cifrado para la transmisión de éstos.

Cifrar la información en su transmisión puede llegar a ser básico en base a las posibilidades que ofrezca el juego que se quiera desarrollar, ya que no sólo dificultará la posibilidad de hacer trampas, sino que será imprescindible en caso de que en el juego se manejen datos reales de información personal como nombres, direcciones, tarjetas de crédito para realizar microtransacciones, etc.

En lo que respecta a cifrado se tienen dos posibilidades, cifrado simétrico y cifrado asimétrico, cada uno con sus ventajas e inconvenientes, a saber:

- **Cifrado simétrico**

El cliente y el servidor comparten una misma clave para cifrar y descifrar los datos transmitidos.

Como ventaja se tiene que este tipo de cifrado es significativamente más rápido que el asimétrico.

En cuanto a desventaja, que introduce la dificultad de cómo intercambiar la clave entre cliente y servidor de forma segura.

Para paliar esta desventaja, se podría generar a través de datos conocidos tanto por el cliente como por el servidor para no tener que transmitirla en texto claro y que pudiera ser interceptada, pero en caso de que un posible atacante fuera capaz de descubrir en que se basan estos datos conocidos por las dos partes implicadas, también podría deducir la clave, capturar y descifrar los paquetes.

- **Cifrado asimétrico**

En este caso tanto el cliente como el servidor tienen dos claves, una clave pública y que es conocida por las dos partes y con la que se cifrará la información que se quiere enviar, y una clave privada conocida únicamente por el destinatario que será con la que se descifre la información. Otra característica de este tipo de cifrado es que la pareja de claves (pública y privada) sólo pueden ser generadas una vez asumiendo así que no puedan ser generadas por otro usuario.

Como ventaja se tiene que este cifrado es mucho más seguro ya que para descifrar la información es necesario conocer la clave privada de una de las dos entidades, y esto sólo serviría para capturar la información en uno de los sentidos de la comunicación.

En cuanto a desventaja, que este tipo de cifrado requiere una mayor capacidad y tiempo de procesamiento que podría introducir un mayor tiempo de respuesta en las comunicaciones.

Presentadas las ventajas e inconvenientes de ambos tipos de cifrado, la solución más eficiente consiste en realizar un híbrido de ambos cifrados del siguiente modo:

- El servidor posee una clave pública y una clave privada, y haciendo uso de cifrado asimétrico el jugador que ejecuta el cliente envíe una clave encriptándola con la clave pública del servidor.
- El servidor desencripta esta clave transmitida con su clave privada, ahora cliente y servidor conocen esta clave y se ha comunicado a través de un canal seguro.
- Cliente y servidor pueden comenzar una comunicación basada en cifrado simétrico con la clave que ambos conocen.

Finalmente cabe indicar que en el caso de la prueba de concepto que acompaña al proyecto no ha sido implementada ninguna de las soluciones de cifrado propuestas debido a que no será transmitida ninguna información de carácter sensible, en cambio para dotar de una mayor estabilidad a las comunicaciones sí será incluido un mecanismo de comprobación de coherencia de los paquetes transmitidos entre cliente y servidor.

4.2.4.Protocolo de transporte, flujo de datos entre servidores y clientes

Como ya se explicó en el punto 4.1.4 Protocolo de transporte, TCP frente a UDP, el protocolo escogido para la comunicación será TCP, tanto en el proceso de login tanto por parte de los servidores como de los clientes como en el proceso de selección de personajes ya que no se puede permitir la pérdida de datos porque producirían estados inconsistentes.

De igual modo se utilizará TCP para la transmisión de la información de juego como son las coordenadas de los jugadores, acciones que realicen, etc. De este modo se garantizará que todos los jugadores tengan la última información del resto de jugadores y que ninguna acción de éstos no sea recibido por los demás, y para paliar la sobrecarga de red producida por este protocolo al tener confirmación de recepción respecto a los movimientos y envío de coordenadas, se enviará un único paquete a cada movimiento de los jugadores indicando la posición en la que se encuentra y la dirección del movimiento, permitiendo con estos datos que el resto de jugadores pueda procesar los movimientos de éste sin necesidad de un flujo constante de información.

A continuación se muestra en mayor detalle los servidores y cliente participantes en la solución propuesta para la infraestructura MMO, descripción de éstos y de sus funcionalidades y el protocolo que seguirán para comunicarse.

- **Servidor 1 - Servidor de conexión SC**

Registra la conexión del resto de servidores del sistema distribuido y la conexión de nuevos clientes. Cuando un cliente se conecte al SC, este le mandará las direcciones de los servidores SL y SPJ si estos se conectaron, o un error indicando que los servidores no están online.

Las distintas negociaciones con los servidores siguen la siguiente secuencia:

- Conexión de un Servidor de Login

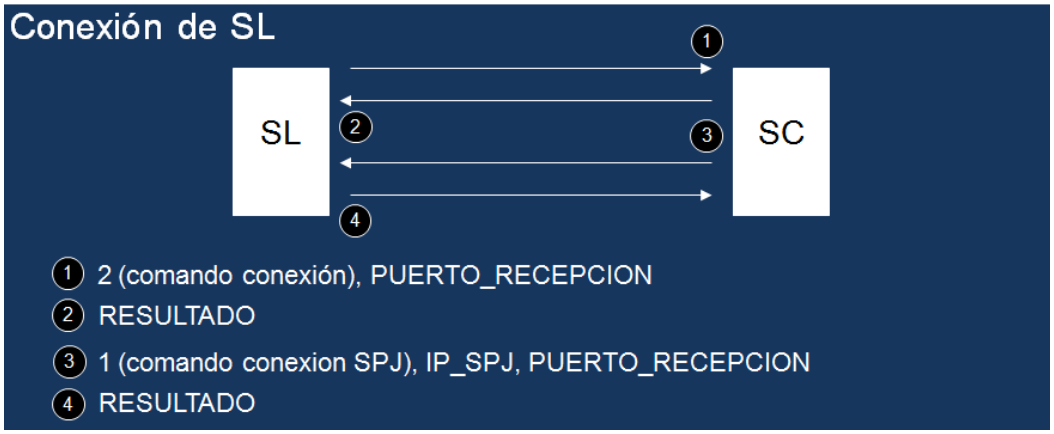

Imagen 17. Protocolo de establecimiento de conexión del Servidor de Login en el sistema

El protocolo que se seguirá para el establecimiento de conexión consiste en que una vez se ha conectado el Servidor de Login al Servidor de Conexión, le envía un paquete de datos que contiene:

- **Comando de conexión**. Un byte con valor 2 que significa que un Servidor de Login quiere darse de alta en el Servidor de Conexión
- **Puerto de recepción**. Tres pares de bytes para indicar el puerto por el que prestará servicio el Servidor de Login

A esta información, el Servidor de Conexión responderá con:

- **Resultado**. Un byte 1 en caso de que pueda registrar ese Servidor de Login, o un byte 0 en caso contrario

A continuación, el Servidor de Conexión enviará al Servidor de Login todos los Servidores de Juego que estén registrados en él:

- **Comando de conexión**. Un byte con valor 1 que significa que un Servidor de Juego quiere darse de alta en el Servidor de Login
- **IP del Servidor de Juego**. Cuatro bytes para indicar la IP del Servidor de Juego
- **Puerto de recepción**. Tres pares de bytes para indicar el puerto por el que prestará servicio el Servidor de Juego

A esta información, el Servidor de Login responderá con:

- **Resultado**. Un byte 1 en caso de que pueda registrar ese Servidor de Juego, o un byte 0 en caso contrario

Finalmente se cerrará la conexión entre ambos servidores

- Desconexión de un Servidor de Login

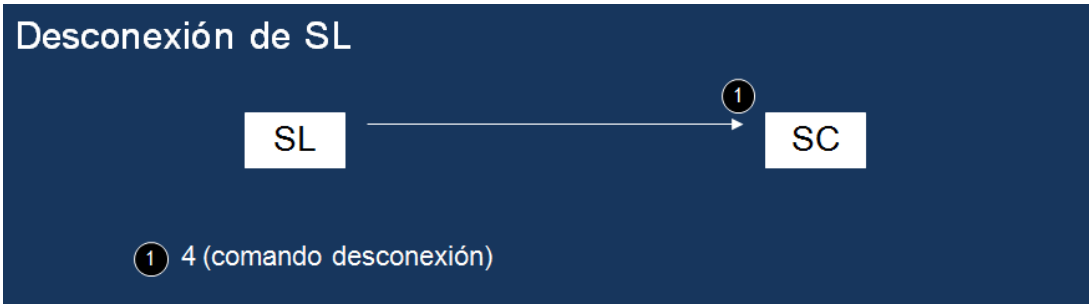

Imagen 18. Protocolo de desconexión del Servidor de Login en el sistema

El protocolo que se seguirá para la desconexión consiste en que una vez se ha conectado el Servidor de Login al Servidor de Conexión, le envía un paquete de datos que contiene:

- **Comando de desconexión**. Un byte con valor 4 que significa que un Servidor de Login quiere darse de baja en el Servidor de Conexión

Finalmente se cerrará la conexión entre ambos servidores.

- Conexión de un Servidor de Juego

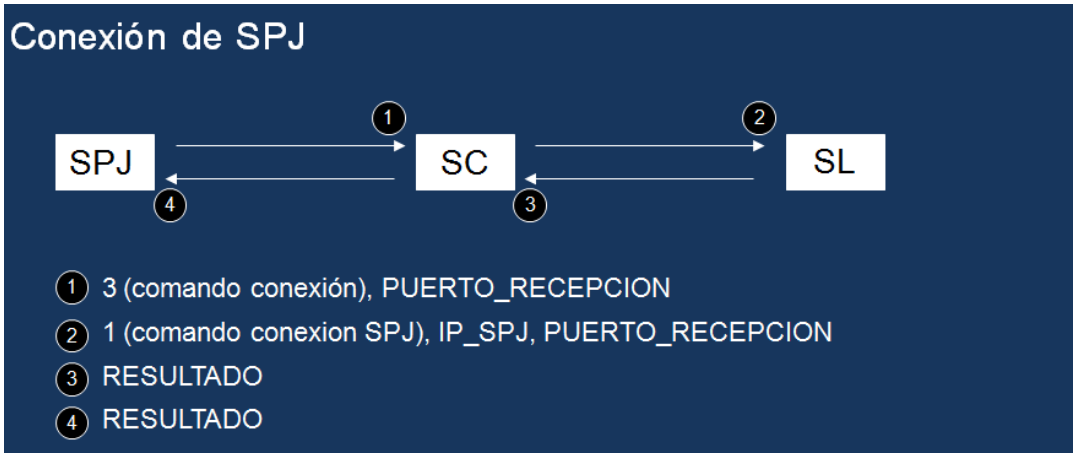

Imagen 19. Protocolo de establecimiento de conexión del Servidor de Juego en el sistema

El protocolo que se seguirá para el establecimiento de conexión consiste en que una vez se ha conectado el Servidor de Juego al Servidor de Conexión, le envía un paquete de datos que contiene:

- **Comando de conexión**. Un byte con valor 3 que significa que un Servidor de Juego quiere darse de alta en el Servidor de Conexión
- **Puerto de recepción**. Tres pares de bytes para indicar el puerto por el que prestará servicio el Servidor de Juego

Recibida esta información, el Servidor de conexión se conectará al Servidor de Login y le enviará un paquete de datos que contiene:

- **Comando de conexión**. Un byte con valor 1 que significa que un Servidor de Juego quiere darse de alta en el Servidor de Login
- **IP del Servidor de Juego**. Cuatro bytes para indicar la IP del Servidor de Juego
- **Puerto de recepción**. Tres pares de bytes para indicar el puerto por el que prestará servicio el Servidor de Juego

El Servidor de Login responderá al Servidor de Conexión con:

- **Resultado**. Un byte 1 en caso de que pueda registrar ese Servidor de Juego, o un byte 0 en caso contrario

Y recibido este resultado, el Servidor de Conexión cerrará la conexión con el Servidor de Login y responderá al Servidor de Juego con:

- **Resultado.** El resultado recibido por el Servidor de Login

Finalmente se cerrará la conexión entre ambos servidores.

Queda estudiar la casuística en que un Servidor de Juego se quiera registrar en el Servidor de Conexión sin que haya un Servidor de Login, en dicho caso el Servidor de

Conexión registrará al Servidor de Juego sin conectar con ningún Servidor de Login, y cuando se registre un Servidor de Login como ya se ha visto le enviará la información con los Servidores de Juego registrados.

- Desconexión de un Servidor de Juego

Imagen 20. Protocolo de desconexión del Servidor de Juego en el sistema

El protocolo que se seguirá para la desconexión consiste en que una vez se ha conectado el Servidor de Juego al Servidor de Conexión, le envía un paquete de datos que contiene:

- **Comando de desconexión**. Un byte con valor 5 que significa que un Servidor de Juego quiere darse de baja en el Servidor de Conexión

Recibida esta información, el Servidor de conexión se conectará al Servidor de Login y le enviará un paquete de datos que contiene:

- **Comando de desconexión**. Un byte con valor 2 que significa que un Servidor de Juego quiere darse de baja en el Servidor de Conexión
- **IP del Servidor de Juego.** Cuatro bytes indicando la IP del Servidor de Juego que quiere darse de baja

Finalmente se cerrará la conexión entre el Servidor de Conexión y el Servidor de Login, y después entre el Servidor de Conexión y el Servidor de Juego.

- Conexión de un Jugador

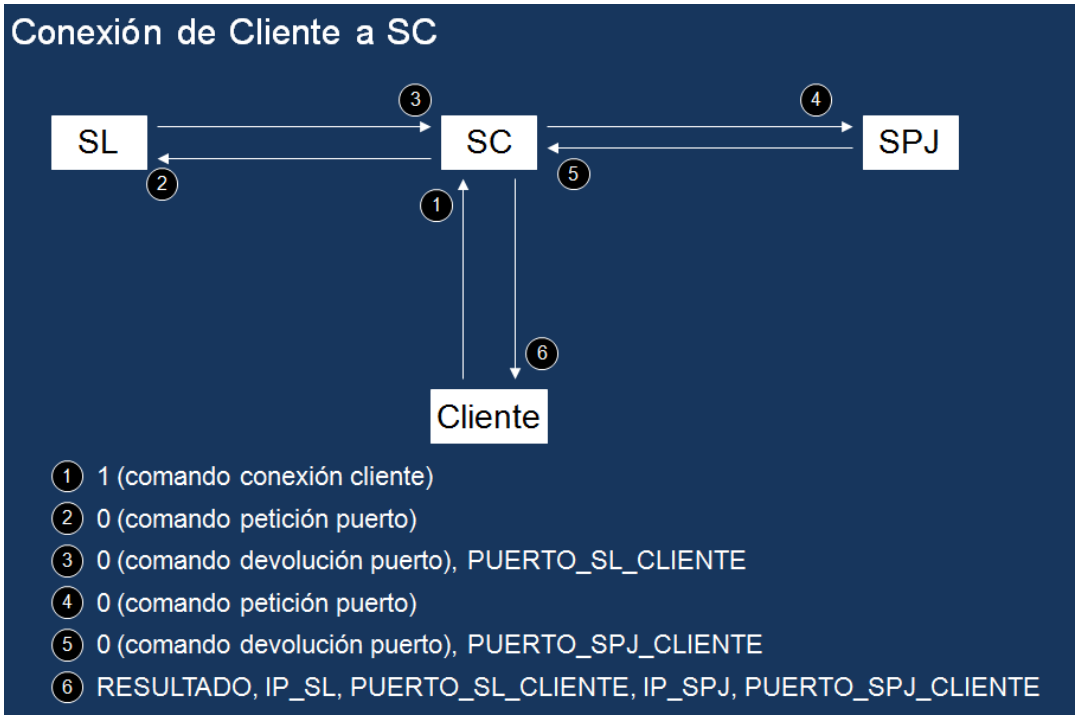

Imagen 21. Protocolo de establecimiento de conexión del Cliente en el sistema

El protocolo que se seguirá para el establecimiento de conexión consiste en que una vez se ha conectado el Jugador al Servidor de Conexión, le envía un paquete de datos que contiene:

- **Comando de conexión**. Un byte con valor 1 que significa que un Jugador quiere darse de alta en el Servidor de Conexión

Recibida esta información, el Servidor de conexión se conectará al Servidor de Login y le enviará un paquete de datos que contiene:

- **Comando de petición de puerto**. Un byte con valor 0 que significa que un Cliente quiere conectarse al Servidor de Login

El Servidor de Login responderá al Servidor de Conexión con:

- **Comando de devolución de puerto**. Un byte con valor 0 que significa que el Servidor de Login ha reservado un puerto de escucha para el Cliente
- **Puerto del Servidor de Login para Cliente**. Tres pares de bytes con el puerto que el Servidor de Login ha reservado para que se conecte el cliente

Recibido este resultado se cierra la conexión entre estos servidores, y a continuación el Servidor de conexión se conectará al Servidor de Juego y le enviará un paquete de datos que contiene:

- **Comando de petición de puerto**. Un byte con valor 0 que significa que un Cliente quiere conectarse al Servidor de Login

El Servidor de Juego responderá al Servidor de Conexión con:

- **Comando de devolución de puerto**. Un byte con valor 0 que significa que el Servidor de Login ha reservado un puerto de escucha para el Cliente
- **Puerto del Servidor de Juego para Cliente**. Tres pares de bytes con el puerto que el Servidor de Juego ha reservado para que se conecte el cliente

Recibida esta información se cierra la conexión entre estos servidores, y a continuación el Servidor de conexión responderá finalmente al cliente con un paquete de datos que contiene:

- **Resultado**. Un byte con valor 1 si están preparados todos los servidores para que se conecte un jugador, 0 en caso de que el jugador no se pueda conectar por problemas de conexión con alguno de los servidores
- **IP del servidor de Login**. IP del servidor de Login que va a atender al jugador
- **Puerto del servidor de Login**. Puerto por el que atenderá el Servidor de Login al jugador
- **IP del servidor de Juego**. IP del servidor de Juego que va a atender al jugador
- **Puerto del servidor de Juego.** Puerto por el que atenderá el Servidor de Juego al jugador

Finalmente se cerrará la conexión entre ambos servidores.

- **Servidor 2 - Servidor de login SL**

Servidor encargado del login del jugador. Una vez el cliente se ha conectado al SC y recibido los servidores que están en marcha, se conectará al SL para ver sus personajes, crear personajes nuevos o borrarlos, etc. Una vez seleccione un personaje, conectará con el servidor SPJ.

Las negociaciones que se realizan en este servidor siguen la siguiente secuencia:

- Conexión y autenticación de jugador

Imagen 22. Protocolo de establecimiento de conexión entre Cliente y Servidor de Login

El protocolo que se seguirá para el establecimiento de conexión consiste en que una vez se ha conectado el Jugador al Servidor de Login, le envía un paquete de datos que contiene:

- **IP del Servidor de juego**. Cuatro bytes indicando la IP del Servidor de Juego que el Servidor de Conexión le ha indicado que le atenderá

Con esta información, el Servidor de Login le responderá:

- **Identificador en Servidor de Login**. Un byte indicando un valor que lo identificará dentro de ese servidor, valor que sumado al puerto por el que atiende al Servidor de Conexión da el puerto por el que atenderá a dicho jugador

Recibida esta información, el jugador enviará el nombre de usuario y contraseña para autenticarse en el sistema:

- **Nombre de usuario.** Array de caracteres indicando el nombre del usuario
- **Contraseña de usuario.** Array de caracteres indicando la contraseña del usuario
-

Si los datos de autenticación no son válidos el Servidor de Login le devolverá un byte 0 para que vuelva a autenticarse, este proceso se repetirá hasta que introduzca unos datos correctos. Una vez que se introducen datos correctos, el Servidor de Login le devuelve al jugador:

- **Comando de confirmación.** Un byte con valor 1 indicando que la autenticación fue correcta
- **Lista de personajes.** Arrays de caracteres indicando los nombres de los personajes con sus características codificadas a nivel de byte

Finalizado este proceso de autenticación y la recepción de personajes y características, comenzaría el proceso de selección, borrado y creación de personajes.

- Selección, borrado y creación de personajes

Imagen 23. Protocolo seguido en las peticiones de servicio entre Cliente y Servidor de Login

Una vez se ha autenticado el jugador en el Servidor de Login, le enviará el siguiente paquete en función de la opción deseada:

- **Identificador en el Servidor de Login**. Un byte indicando el identificador que le adjudicó el Servidor de Login al inicio del proceso de autenticación
- **Comando**. Un byte indicando el comando de la función. Su valor será 0 para la creación de un nuevo personaje, 1 para la selección de un personaje existente y 2 para el borrado de un personaje existente
- **Nombre del personaje**. Sea cual sea el comando mandado, se enviará el nombre del personaje como un array de caracteres. En el caso de la creación incluirá codificado en bytes las características del personaje, para los casos de selección y borrado envía el nombre del personaje para identificar el personaje a borrar

Con esta información, el Servidor de Login le responderá:

- **Resultado**. Un byte con valor 1 si todo ha ido correctamente o 0 en caso de error.

En el caso de que el jugador envíe un comando de selección de personaje, el Servidor de Login se pondrá en contacto con el Servidor de Juego para informarle del personaje y características de éste.

- Selección de un personaje

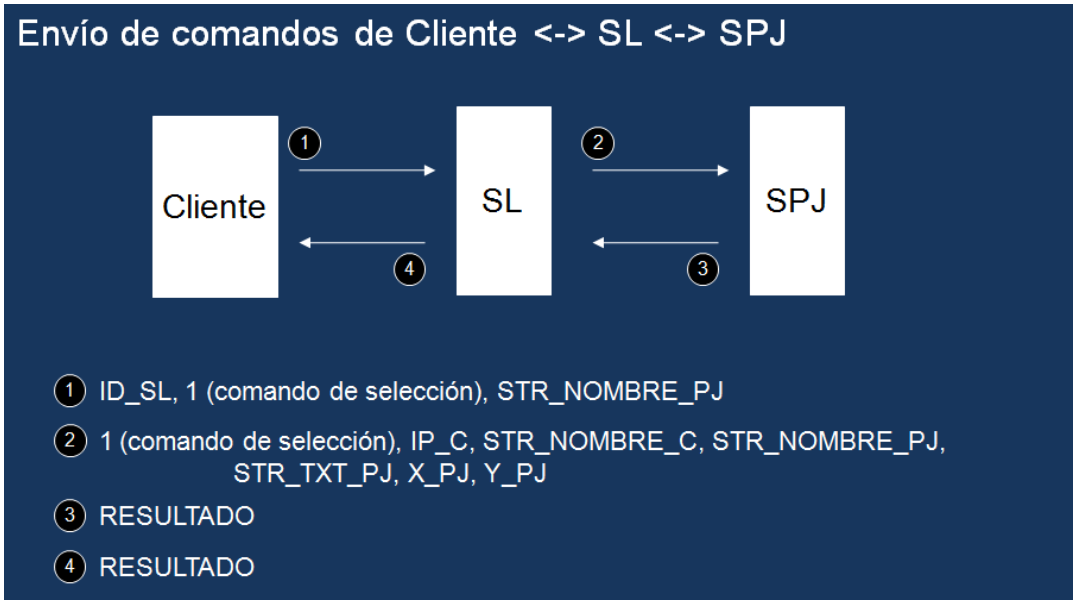

Imagen 24. Protocolo seguido en la selección del personaje de juego

El protocolo que se seguirá para la selección del personaje comenzará con el envío del comando por parte del jugador al Servidor de Login como se ha explicado en el punto anterior:

- **Identificador en el Servidor de Login**. Un byte indicando el identificador que le adjudicó el Servidor de Login al inicio del proceso de autenticación
- **Comando de selección**. Un byte con valor 1 indicando que se ha seleccionado un personaje
- **Nombre del personaje.** Se enviará el nombre del personaje como un array de caracteres.

Con esta información, el Servidor de Login conectará con el Servidor de Juego y le enviará la siguiente información:

- **Comando de selección**. Un byte con valor 1 indicando que el jugador ha seleccionado un personaje
- **IP del jugador**. Cuatro bytes indicando la IP del jugador
- **Nombre de usuario del jugador**. Array de caracteres con el nombre de usuario del jugador
- **Nombre del personaje del jugador**. Array de caracteres con el nombre del personaje seleccionado del jugador más las características codificadas a nivel de byte
- **Textura del personaje**. Array de caracteres con el nombre de la textura que se usará para modelar al jugador
- **Coordenada X del personaje**. Dos bytes para indicar la coordenada X del personaje seleccionado por el jugador

- **Coordenada Y del personaje**. Dos bytes para indicar la coordenada Y del personaje seleccionado por el jugador

Recibida esta información, el Servidor de Juego responderá al Servidor de Login:

- **Resultado.** Un byte con valor 1 si todo fue correctamente y un byte 0 en caso de error

Una vez tenga este resultado el Servidor de Login, se lo transmitirá al jugador para que en caso correcto se negocie el cierre de conexión y pase al Servidor de Juego.

- **Servidor 3 - Servidor de juego SPJ**

Servidor de juego con el que el cliente y los personajes no jugadores se conectarán para verificar toda la lógica del juego (movimientos, interacción entre usuarios, etc.).

Las negociaciones que se realizan en este servidor siguen la siguiente secuencia:

- Conexión en dirección Cliente-Servidor

Imagen 25. Protocolo de establecimiento de conexión entre Cliente y Servidor de Juego

El protocolo que se seguirá para el establecimiento de conexión dirección Cliente-Servidor y la cual servirá para que éste le envíe su estado consistirá en la conexión por parte del Cliente al Servidor de Juego:

- **Vacio**. Únicamente establecerá conexión pero no mandará ningún dato

Tras recibir la conexión, el Servidor de Juego le responderá:

- **Identificador en el Servidor de Juego**. Un byte indicando un valor que lo identificará dentro de ese servidor, valor que sumado al puerto por el que atiende al Servidor de Juego da el puerto por el que atenderá a dicho jugador

Terminada esta secuencia se establecerá la conexión en el sentido contrario.

- Conexión en dirección Servidor-Cliente

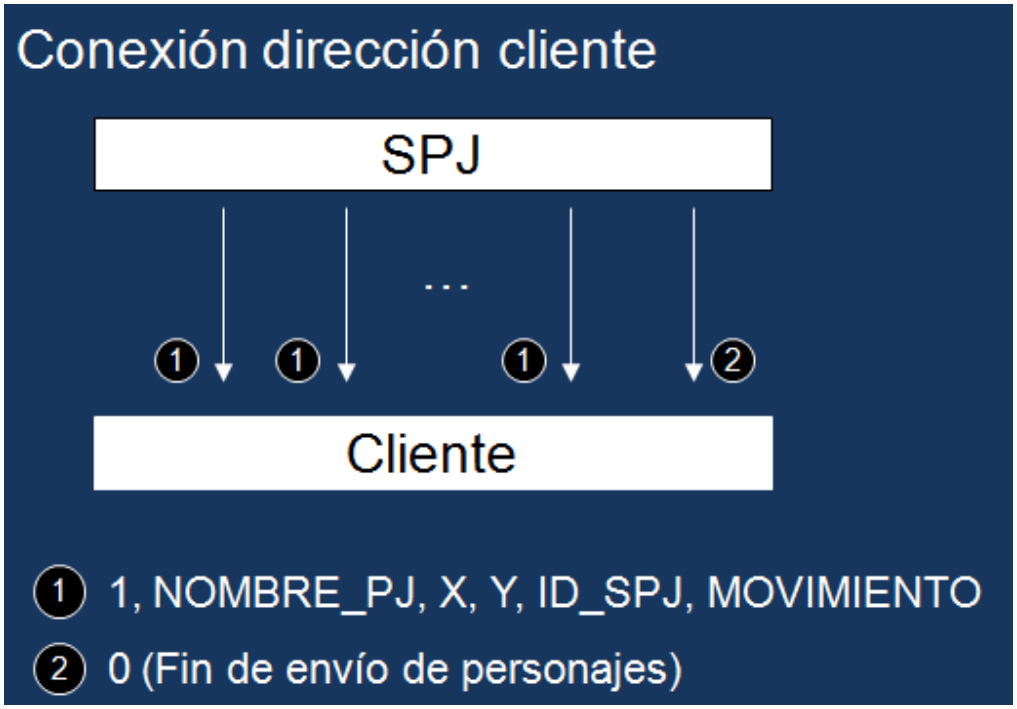

Imagen 26. Protocolo de establecimiento de conexión entre Servidor de Juego y Cliente

El protocolo que se seguirá para el establecimiento de conexión dirección Servidor-Cliente y que servirá para mantenerle actualizado con la información del resto de jugadores consistirá en el envío por parte del Servidor de Juego al Cliente de un paquete con la siguiente información:

- **Comando de conexión de jugador**. Un byte con valor 1 que indica que un jugador nuevo se ha conectado a la partida
- **Nombre del personaje**. Un array de caracteres con el nombre del personaje que se ha conectado
- **Coordenada X**. Dos bytes para indicar la coordenada X del jugador
- **Coordenada Y**. Dos bytes para indicar la coordenada Y del jugador
- **Identificador del Servidor de Juego**. Identificador asignado por el servidor de juego para dicho jugador
- **Acción**. Un byte con un valor entre 0 y 15 para indicar el movimiento o acción que está realizando el personaje en este momento

Al tratarse del momento de inicio de la partida, realizará este paso por cada uno de los jugadores que estén conectados actualmente.

Para indicar al jugador que no hay más jugadores conectados de los que informar, enviará el siguiente paquete:

- **Comando de comienzo de diálogo**. Un byte con valor 0 que indica que ya no hay más jugadores conectados de los que informar, y que puede empezar el diálogo

Terminado de recibir todos los jugadores, comenzará la secuencia de envío/recepción de datos para mantener al servidor y al jugador actualizado en todo momento.

- Diálogo entre Servidor y Cliente

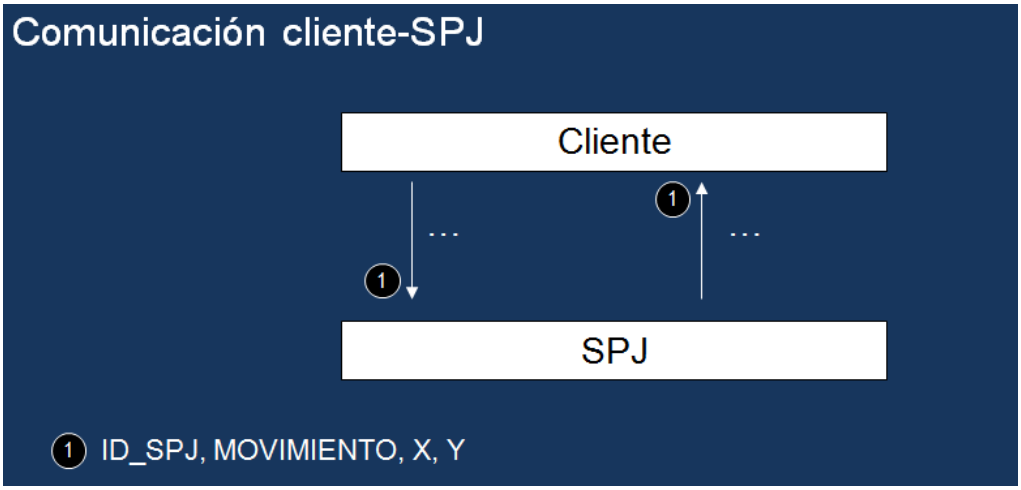

Imagen 27. Protocolo de comunicación de datos entre Servidor de Juego y Cliente

Una vez que se ha establecido la conexión en ambas direcciones comienza el diálogo entre el cliente y el servidor, los cuales se mandarán paquetes con la siguiente información:

- **Identificador del Servidor de Juego**. Identificador asignado por el servidor de juego para dicho jugador
- **Acción**. Un byte con un valor entre 0 y 15 para indicar el movimiento o acción que está realizando el personaje en este momento
- **Coordenada X**. Dos bytes para indicar la coordenada X del jugador
- **Coordenada Y**. Dos bytes para indicar la coordenada Y del jugador

Este paquete será mandado por el jugador hacia el Servidor de Juego cada vez que realice una nueva acción y cada X milisegundos como se explicará más adelante para mantener.

A su vez, por cada uno de estos paquetes que reciba el Servidor de Juego, comprobará que jugadores pueden verse afectados en función de la distancia entre personajes o tipo de acción y la reenviará a éstos para que tengan información del evento.

- **Cliente**

Puesto que todas las comunicaciones realizadas por el cliente tienen como objetivo la transmisión de datos entre éste y el servidor, todas las negociaciones que el cliente realiza son las tratadas en las explicaciones anteriores.

4.2.5.Protocolo de transporte, frecuencia en el envío de paquetes

Otro problema inherente a los juegos de tipo multijugador es la frecuencia en el envío de paquetes, problema aun más importante si se trata de un MMO donde se tiene que llegar a dar soporte a una alta cantidad de jugadores a la vez.

A la hora de resolver este problema se plantean distintos métodos:

- **Reducir la necesidad de transmisión de datos en el juego**

Dado que se trata de un juego de tipo social en el que la actividad de los jugadores tendrá un ritmo más pausado que en otros estilos de juegos, el sistema de acciones de los jugadores se basará en los eventos producidos de la interacción del jugador.

De este modo se propone que en lugar de transmitir de forma continua las coordenadas del jugador, se transmitan únicamente los cambios en las acciones de éste, por ejemplo:

- El jugador está quieto. Se enviará un único paquete que indicará a los jugadores que está quieto en una determinada posición (coordenadas X e Y)
- El jugador comienza a moverse hacia arriba. Se enviará un único paquete indicando la dirección del movimiento y las coordenadas en que se origina este movimiento.
- El jugador deja de moverse hacia arriba y se detiene. Se enviará un paquete indicando que está quieto en la posición que se encuentre.

Siguiendo esta idea se garantiza un envío minimizado de paquetes que se reducirá estrictamente a las acciones realizadas por los jugadores, sin saturar la conexión del cliente y, más importante, del Servidor de Juego que será quien reciba los movimientos de todos los jugadores.

- **Reducir la retransmisión de datos innecesaria desde el servidor**

En este sentido, es innecesario que un servidor retransmita datos entre jugadores cuyas acciones no tengan repercusión entre ellos.

Siendo el caso por ejemplo de dos jugadores que se encuentren a suficiente distancia como para que tras realizar sus movimientos no entren uno en el rango de visión del otro. Ese tiempo de transferencia ahorrado cuando se trata de un MMO puede alcanzar cotas suficientemente elevadas como para ser más eficiente la introducción de código que controle la necesidad de transmitir o no la información.

Aplicadas estas dos posibilidades se tendría un sistema de comunicaciones muy eficiente. No obstante no hay que olvidar que se trata de transmisión de datos en la red y en estas transmisiones se introducirán tiempos de latencia ya que las acciones y coordenadas no llegarán en el mismo instante de transmisión y por tanto se puede tener información desfasada de los jugadores.

Para paliar este efecto y corregir desfases de cálculos de coordenadas entre movimientos en las distintas aplicaciones cliente debido al retardo producido en la transmisión de datos en una red, se propone:

- Enviar paquetes sólo cuando estos aporten información nueva.
- Enviar un número de paquetes razonable que no sature la comunicación con el servidor

Del primer punto se puede establecer que no sería necesario enviar información si el jugador está quieto, ya que no está variando su estado y si se enviara información se estaría dedicando un ancho de banda innecesario a éste.

Del segundo punto se establece que un número elevado de paquetes por segundo llevaría a una pérdida de eficiencia significativa por parte del servidor ya que habría que dedicar mucho tiempo a la gestión de retransmisión de mucha información por parte de cada jugador, y este tiempo se podría ver incrementado en caso de que se introdujera lógica que controlara de algún modo esta información como se comentó anteriormente. Por ello y dado que el tipo de juego es de ejecución pausada por parte de los jugadores, se propone que en caso de movimientos de los jugadores se envíen de 2 a 4 paquetes por segundo.

De este modo y resumiendo el apartado, se propone como solución eficiente:

- El envío de datos cuando el jugador realice un cambio de acción respecto a su acción anterior, indicando el movimiento y coordenada en que lo realiza.
- El envío de 2 a 4 paquetes por segundo si el jugador está en movimiento, indicando el movimiento que realiza y la coordenada actual.
- Introducción de lógica para controlar a quién sí y a quién no retransmitir la información recibida de un jugador.

4.2.6. Protocolo de transporte, ajuste de coordenadas por latencia

Como se indicó en el apartado anterior, al transmitir información a través de la red se produce una latencia que consiste en un tiempo de retardo entre que se envía la información desde un nodo del sistema a otro, aplicado al caso que se está tratando, en el envío de información de un jugador hasta otro jugador. Debido a esto el jugador local puede estar representando gráficamente a jugadores remotos en coordenadas que no están ocupando, y esta representación será más inexacta cuanta mayor latencia se introduzca en el sistema.

Adicionalmente, al retardo introducido por la latencia hay que sumar el tiempo de proceso dedicado al procesamiento por parte del servidor de la información como por ejemplo la comprobación de que la estructura de datos recibida es correcta, comprobación de a qué jugadores se enviará, retardo introducido por la latencia del servidor al resto de jugadores, etc.

Por todos estos motivos, la solución que se ha propuesto evita basarse en el envío de las coordenadas del jugador para su representación en el instante ya que sería imposible mantener un estado real en la aplicación cliente ejecutada por todos los jugadores sin saturar con estas transmisiones al Servidor de Juego.

En su lugar y como se ha explicado en apartados anteriores se enviarán los movimientos y coordenadas donde se originan estos movimientos para que los jugadores al recibir este paquete puedan predecir en cualquier momento el movimiento que está realizando un jugador remoto y en caso de error de cálculo reajustar las coordenadas.

De modo que la resolución del sistema se ajusta el siguiente ejemplo:

- Un jugador remoto presiona la tecla “arriba” y comienza a moverse hacia arriba, por lo que se enviará un paquete con el movimiento y las coordenadas en las que se encuentra al Servidor de Juego que reenviará esta información al jugador local.
- El jugador local recibe el movimiento del jugador remoto, ahora sabe que en ese instante se encuentra en unas coordenadas determinadas y se está moviendo hacia arriba, guardará junto con esta información el instante en que se ha recibido el paquete.
- Mientras no reciba nueva información del jugador remoto, el jugador local pintará a ese jugador en la posición que recibió más las coordenadas calculadas en función del movimiento y de la velocidad de desplazamiento del jugador (que para todos será la misma) en esa dirección.
- El jugador remoto suelta la tecla “arriba” y se queda quieto, por lo que se envía un paquete indicando que se detiene y la coordenada en la que se ha detenido al Servidor de Juego que reenviará esta información al jugador local.
- El jugador local recibe el movimiento del jugador remoto, así que deja de pintar el movimiento hacia arriba y lo deja quieto en la coordenada recibida.

En este ejemplo se puede apreciar el problema introducido por el retardo en la recepción de la información al calcular las coordenadas, en el momento en el que el jugador local recibe la información de que el jugador remoto se ha detenido en una determinada

coordenada puede surgir la diferencia de coordenadas, ya que si este paquete llega con retardo la aplicación cliente del jugador local habrá pintado al jugador remoto más adelantado de la posición en la que se ha quedado quieto, y al recibir esta información y ajustar las coordenadas del jugador remoto se realizará un "salto hacia atrás" en el gráfico que represente a este jugador remoto.

No obstante se puede reducir el impacto del problema presentado a través de las siguientes alternativas:

- **Corrección progresiva de las coordenadas**

Una solución más estética que realizar un ajuste de las coordenadas del jugador remoto al recibir la información sobrescribiendo la posición calculada por el jugador local y produciéndose la sensación de que se pega un "salto hacia atrás", sería que si hay diferencia entre la posición del jugador remoto y la calculada por el jugador local, en lugar de realizar un salto, realizara un movimiento dirigido por la aplicación cliente (el juego) hasta colocar al jugador en la posición que realmente está obteniéndose así una ejecución más fluida.

  - Ventajas

    - Es una solución global que siempre resolverá el problema introducido por los retardos.

  - Inconvenientes:

    - Los constantes desajustes debidos al retardo harán que la aplicación cliente tenga que corregir los movimientos con contra-movimientos que a la larga pueden no presentarse tan estéticos.

- **Reducción de la velocidad de movimientos de los jugadores**

Otra posibilidad consiste en reducir la velocidad a la que se mueven los jugadores cuando se trata de un jugador remoto, de este modo cuando se reciba el paquete con el nuevo movimiento puede no haber sobrepasado la coordenada que realmente ocupa. Para que este método tuviera un resultado más exacto, sería interesante que calculara la velocidad de movimiento de los jugadores en base a las latencias de éstos, es decir, que conocieran el retardo medio que están llevando estos jugadores para ajustar la velocidad a la que se mueven y poder prever el punto en el que se encontrará un jugador remoto cuando se reciba un nuevo paquete a través de conocer el retardo que se va a producir.

  - Ventajas

    - Ofrece una solución más específica y que se ajustará al retardo de cada jugador

- Bien realizado el cálculo puede conseguir que no sea necesario un reajuste adicional de coordenadas

- Inconvenientes

  - Introduce mayor complejidad en la programación de la aplicación cliente
  - Introduce mayor complejidad en el protocolo de transmisión definido, ya que también será necesario calcular y transmitir retardos
  - Pese a que el retardo se mantenga de forma general en un tiempo estable, puede surgir que en un determinado paquete se produzca de forma eventual un retardo mucho mayor y que el cálculo estimado de las posiciones no sea válido

De las ventajas e inconvenientes presentados la solución que mejor respuesta daría sería complementar las deficiencias de una opción con la otra implementando ambas.

Dada la complejidad que implementar un sistema eficiente de ajuste de coordenadas y que este estudio podría ser tema de amplia discusión, para el caso práctico desarrollado no serán implementados estos métodos y en su lugar se realizará un cálculo sencillo basado en una velocidad de desplazamiento fijo cada X frames de imágenes.

4.2.7.Concurrencia

A continuación se describe la forma en que serán implementadas las exclusiones mutuas en los diferentes servidores y la aplicación cliente para garantizar que la concurrencia en el acceso a los datos no genere estados inconsistentes de información.

- **Servidor de Login**

En el caso del Servidor de Login se tienen múltiples hilos en ejecución de forma simultánea, los cuales acceden a una misma variable que contiene una lista con los jugadores conectados.

Los puntos en que es necesario controlar la concurrencia son:

- Dar de alta a un jugador en el Servidor de Login
- Verificar si el jugador está dado de alta en el Servidor de Login

Para que esta variable tenga el mismo valor en ambos hilos tiene que ser un miembro estático por lo que tendrá el modificador **static** en su declaración:

```
static List entrada;
```

De este modo garantizamos que en todos los hilos de ejecución tendremos acceso a la misma variable, y es en ésta donde tendremos que definir un control para la exclusión mutua en su acceso desde los distintos hilos.

El uso que se le dará será:

- Hilo principal: Inscribirá en esta lista al jugador cuando se conecte al Servidor de Login
- Hilo del cliente: Verificará que el cliente que se conecta está en esta lista, y lo eliminará de la lista finalizada la interacción entre ambos.

El método que se empleará para garantizar la exclusión mutua será la declaración de secciones críticas de código a través de la palabra reservada del compilador de C#, **lock,** y la creación de una variable global y estática que haga de mutex.

```
static object mutex;
```

Una vez que se tiene esta variable, cada vez que se quiera realizar cambios en la lista de jugadores conectados bloquearemos el mutex asegurando que sólo uno de los hilos tendrá acceso a esa sección de código:

```
lock (mutex)
{
    //Realizar cambios en la variable compartida entrada
}
```

- **Servidor de Juego**

En el caso del Servidor de Juego se tienen múltiples hilos en ejecución de forma simultánea y en los cuales se acceden a múltiples variables compartidas.

Los puntos en que es necesario controlar la concurrencia son:

- Negociación de la conexión de un jugador en el Servidor de juego
- Verificar que el cliente que manda comandos está en la lista de jugadores conectados
- Envío de estado de un jugador al resto de jugadores.

Al igual que en el caso anterior, las variables que serán accedidas desde los múltiples hilos y en las que se quiere tener los mismos valores serán definidas como miembros estáticos.

En este caso se tendrá tres listas, una con la información de los clientes que están en proceso de conexión, otra con la información de entrada de datos del cliente, y una última con la información de salida de datos al cliente:

```
static List conectando;
static List entrada;
static List salida;
```

El uso que harán los distintos hilos de estas variables compartidas es:

- Hilo principal: Negociará la conexión de los jugadores en su entrada al Servidor de Juego, inscribiendo al jugador en la lista **conectando** mientras establece una conexión en sentido cliente-Servidor de Juego. Establecida esta conexión se introduce al jugador en la lista **entrada** y se comienza a establecer la conexión en sentido Servidor de Juego-cliente. Terminado de establecer esta segunda conexión se introduce al jugador en la lista **salida** y se le elimina de la lista **conectando**.

- Hilo del cliente: Verificará que el jugador que manda comandos y acciones está en la lista **entrada** y enviará estos comandos y acciones a todos los jugadores que están en la lista **salida**.

Se utilizará el mismo método para garantizar la exclusión mutua, es decir, se creará una variable mutex para evitar el acceso simultáneo a las variables compartidas.

```
static object mutex;
```

De modo que cada vez que se quiera acceder a alguna de esas tres listas se rodeará el código con la instrucción **lock** asegurando el acceso exclusivo a esa sección del código:

```
lock (mutex)
{
    //Realizar cambios en la variable compartida conectando, entrada y/o salida
}
```

- **Cliente**

En el caso del Servidor del Cliente se tienen tres hilos en ejecución de forma simultánea y en los cuales se acceden a múltiples variables compartidas.

Los puntos en que es necesario controlar la concurrencia son:

- Agregar un comando o acción realizado por el jugador
- Envío del comando o acción realizada por el jugador
- Actualización de los datos de un jugador remoto por la recepción de su nuevo estado
- Cálculo de las coordenadas de los jugadores

Las variables compartidas entre múltiples hilos y en las que se quiere tener los mismos valores serán definidas como miembros estáticos. En este caso se tendrán 2 listas, una con la información de los comandos y acciones realizados por el jugador local y otra con el estado de los jugadores (remotos y local).

```
static List comandos;
static List jugadores;
```

El uso que harán los distintos hilos de estas variables compartidas es:

- Hilo encargado de la lógica y puesta en pantalla de los gráficos: Cada vez que actualiza la lógica en pantalla antes de representarla gráficamente, calcula las coordenadas de la lista **jugadores** según los el último movimiento que esté realizando y que ha sido indicado por el Servidor de Juego. También tiene que ser controlado cada movimiento nuevo que realiza el jugador que se agregará a la lista **comandos**.
- Hilo de envío de datos: Envía los comandos y acciones realizados por el jugador local que se encuentran en la lista **comandos**
- Hilo de recepción de datos: Actualiza la información de un jugador remoto con su nuevo movimiento y coordenadas en la lista **jugadores**.

Para garantizar la exclusión mutua en el caso de la actualización de la lista **comandos** y en su envío se creará una variable mutex con el fin de evitar el acceso simultáneo a las variables compartidas.

```
static object mutex;
```

De modo que cada vez que se quiera acceder a alguna de esas tres listas se rodeará el código con la instrucción **lock** asegurando el acceso exclusivo a esa sección del código:

```
lock (mutex)
{
    //Realizar cambios en la variable compartida conectando, entrada y/o salida
}
```

Para el caso del cálculo de las coordenadas de un jugador y la actualización desde el Servidor de Juego se hará uso de una funcionalidad que nos proporciona la librería .NET que consiste en usar el mismo objeto instanciado, como mutex con el fin de crear una sección crítica de código respecto a él.

En este caso como el código que debe ser modificado gira en torno a un único jugador, realizaremos la exclusión mutua del siguiente modo:

```
for (int i = 0; I < jugadores.Count ; i++)
{
    Jugador aux = jugadores[i];
    lock (aux)
    {
        //Actualizamos los datos del jugador
    }
}
```

4.3.Integración en XNA

En los siguientes apartados se realizará una descripción ampliada de la biblioteca XNA de Microsoft para el desarrollo de videojuegos desde una perspectiva técnica.

También se expondrán los pasos necesarios para la creación de un proyecto en esta biblioteca indicando desde los requisitos de instalación hasta la creación, compilación y ejecución de un primer proyecto en blanco.

Sentadas las bases de la creación del proyecto se expondrá de qué partes consta el ciclo de ejecución de XNA y el objetivo de cada una de éstas.

Finalmente se propondrá una posible implementación de la infraestructura MMO expuesta en los apartado 4.1 y 4.2.

4.3.1.Descripción de XNA

Como se exponía en el apartado 3.3.3 Resumen de librerías y motores de juego más populares del mercado, la librería de desarrollo XNA es la plataforma de desarrollo de videojuegos propuesta por Microsoft para la producción de juegos en las distintas plataformas de Windows (sistemas operativos de sobremesa, dispositivos móviles de telefonía y Xbox360).

En su última versión, esta librería de desarrollo se integrará en el IDE Microsoft Visual Studio 2008 permitiendo el desarrollo de juegos en el lenguaje de programación C#.

Los requisitos que el uso de esta plataforma por parte del desarrollador y del usuario final supone son los siguientes:

- **Requisitos de instalación para desarrollar de juegos XNA**

Para el desarrollo utilizando esta biblioteca es necesario descargar el Microsoft XNA Game Studio[36], cuya última versión liberada a fecha de 14 de Febrero de 2010 es la 3.1.

Los requisitos mínimos para hacer uso de este framework son:

- Sistema operativo Windows Vista Service Pack 1 o Windows XP Service Pack 3
- Tarjeta gráfica con soporte para DirectX 9.0c y Shader Model 1.1 (recomendado Shader Model 2.0)
- DirectX 9.0c
- .NET Framework 3.5
- Microsoft Visual C# 2008 Express Edition o Microsoft Visual Studio 2008 Standard Edition

- **Requisitos de instalación para ejecución de juegos XNA**

Para la ejecución de juegos desarrollados en XNA en su versión 3.1, es necesario tener instalado el Microsoft XNA Framework Redistributable 3.1.

Los requisitos mínimos para hacer uso de este framework son:

- Sistema operativo Windows Vista Service Pack 1 o Windows XP Service Pack 3
- Tarjeta gráfica con soporte para DirectX 9.0c y Shader Model 1.1 (recomendado Shader Model 2.0)
- DirectX 9.0c
- .NET Framework 3.5

4.3.2. Pasos para crear un proyecto de juego bajo XNA

A continuación se expone una guía detallada para la instalación y primeros pasos a la hora de crear, compilar y ejecutar un proyecto de juego en XNA.

- **Instalación para desarrollo de juegos en XNA**

Presumiendo instalados los requisitos mínimos (sistema operativo, .NET Framework 3.5 y Microsoft Visual Studio 2008 Standard Edition), los pasos para la instalación del Microsoft XNA Game Studio 3.1 son:

1. Desinstalar versiones anteriores de XNA Game Studio si las hay.
2. Instalar Microsoft Visual C# 2008 Express Edition o Microsoft Visual Studio 2008 Standard Edition o versión superior.
3. Obtener las últimas actualizaciones para Visual Studio desde Microsoft Update.
4. Descargar y ejecutar el instalador de Microsoft XNA Game Studio 3.1.
5. Seguir las instrucciones en pantalla para la instalación.

- **Creación de un proyecto**

Para la creación de un proyecto de juego en XNA, se podrá o bien ejecutar el acceso directo que ya se tuviera de Visual Studio 2008, o ir en el Menú Inicio al directorio Microsoft XNA Game Studio 3.1 y ejecutar el acceso directo Visual Studio 2008 que se ha creado.

Tras ejecutar uno de los accesos directos indicados tendremos la siguiente pantalla:

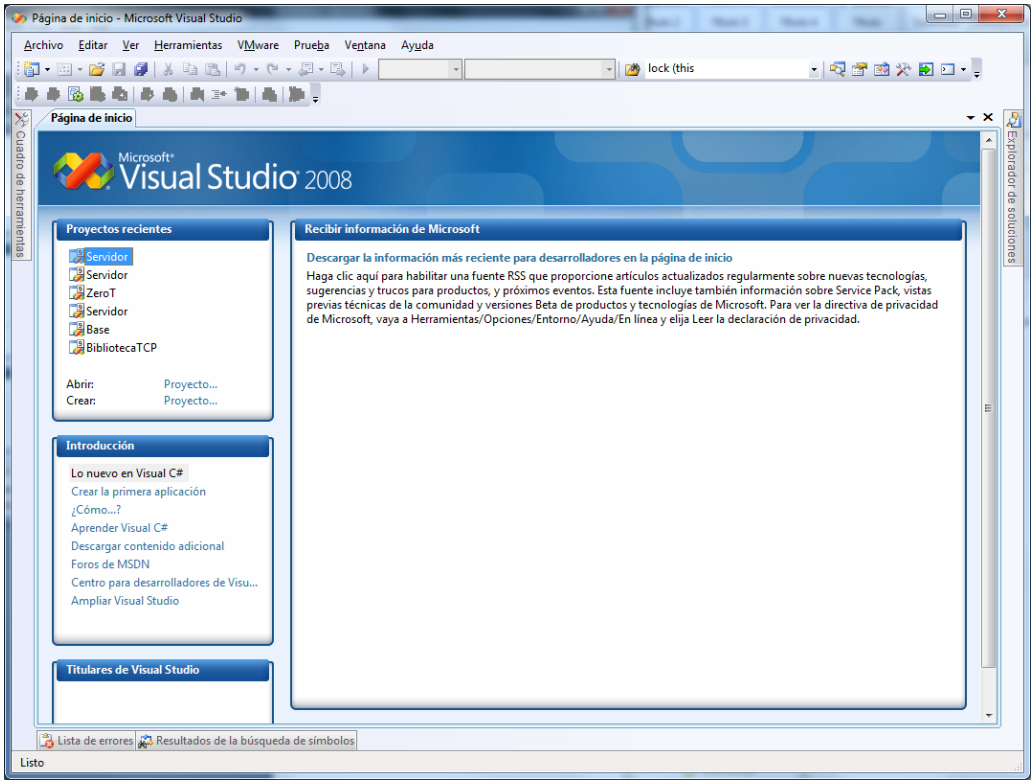

Imagen 28. Página de inicio de Microsoft Visual Studio 2008

Seleccionaremos la opción del menú **Archivo > Nuevo > Proyecto…** que nos mostrará la siguiente captura:

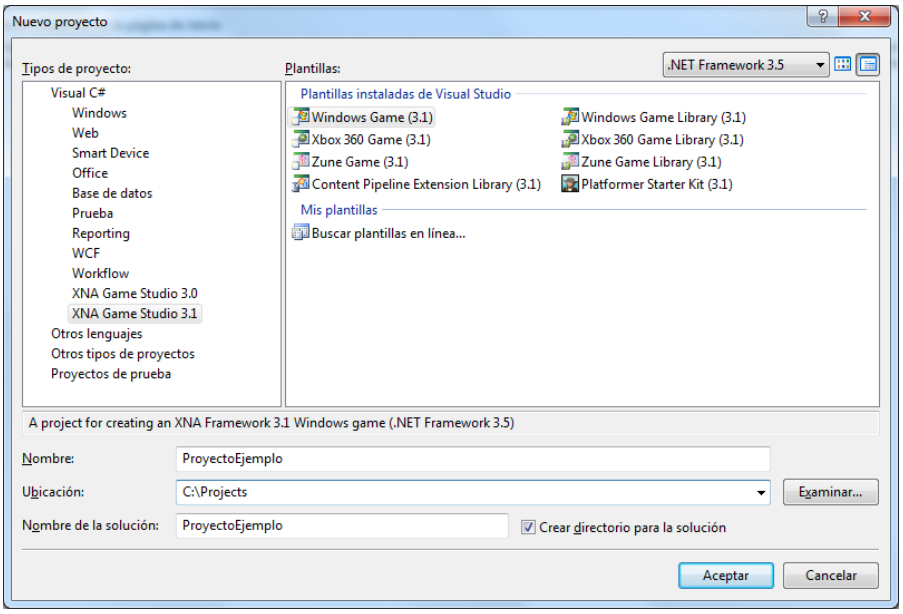

Imagen 29. Creación de un nuevo proyecto en Microsoft Visual Studio 2008

Seleccionaremos la opción Windows Game (3.1), le daremos un nombre al proyecto y presionaremos el botón Aceptar.

Ya tenemos creado el proyecto de juego en XNA, y nos dejará ver la siguiente captura:

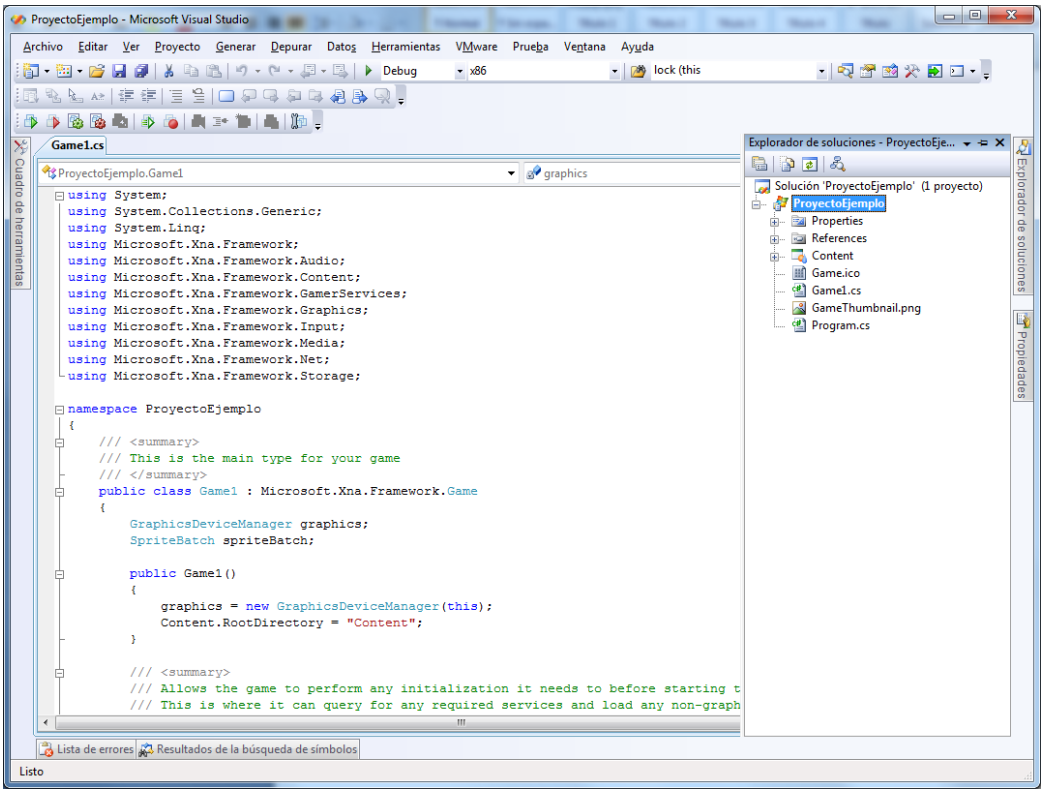

Imagen 30. Captura de pantalla de un proyecto de juego XNA en Microsoft Visual Studio

En este punto ya se puede compilar y ejecutar el juego siguiendo los siguientes pasos:

- Para compilar seleccionaremos en el menú **Generar > Generar Solución**, o presionaremos la tecla **F6**.
- Para ejecutar en modo depuración seleccionaremos en el menú **Depurar > Iniciar depuración**, o presionaremos la tecla **F5.**
- Para ejecutar sin modo depuración seleccionaremos en el menú **Depurar > Iniciar sin depurar**, o presionaremos las teclas **Control+F5.**

Dado que no se ha cargado contenido ni introducido ningún código adicional en el proyecto si compilamos y ejecutamos se tendrá una pantalla con fondo azul.

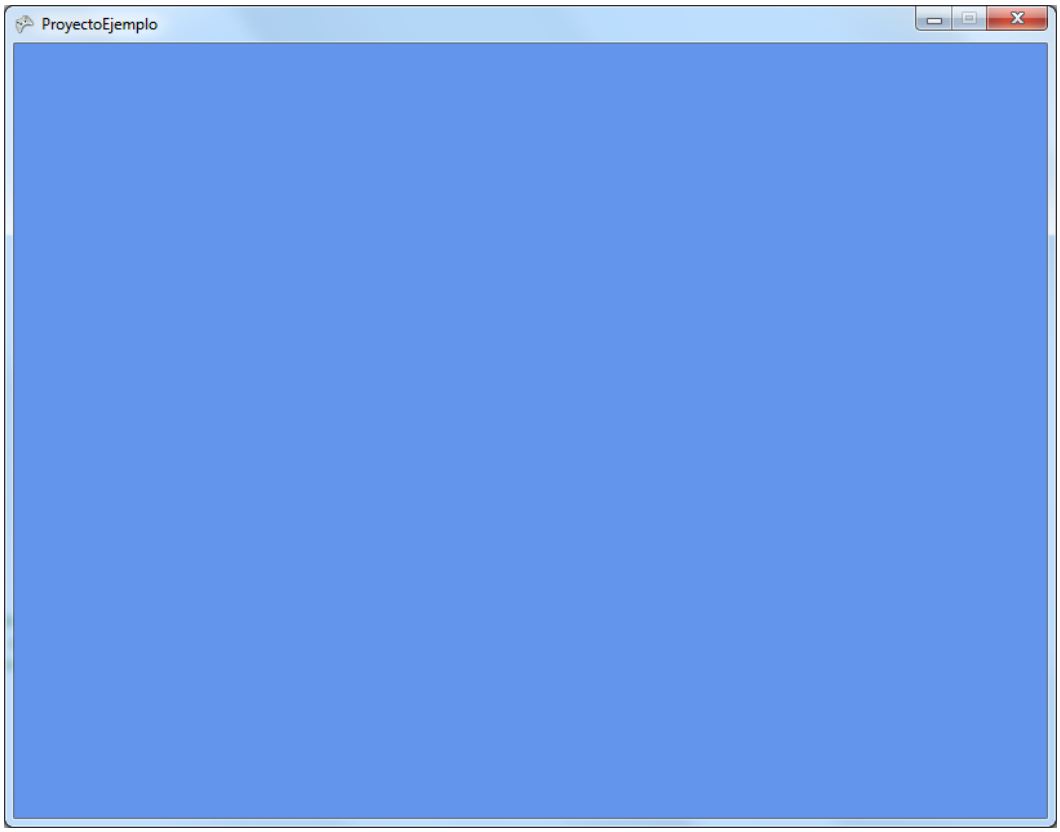

Imagen 31. Proyecto de juego XNA en Microsoft Visual Studio 2008

4.3.3.Resumen descriptivo del ciclo de ejecución de un juego en XNA

En este apartado se abordará el ciclo de ejecución de una aplicación XNA partiendo del código generado en la creación de un proyecto nuevo, para ello se abre el proyecto generado en el apartado anterior.

Si se observa el apartado del Explorador de soluciones (en la barra de menú de Visual Studio seleccionar **Ver > Explorador de Soluciones**) se tendrá la siguiente captura:

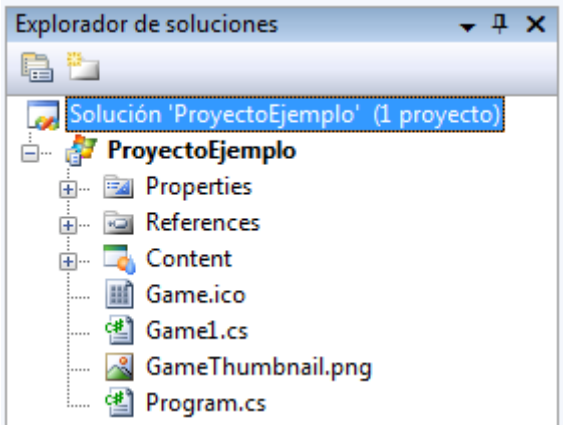

Imagen 32. Explorador de soluciones de Microsoft Visual Studio 2008

La clase Program.cs contendrá el punto de inicio del juego:

```
using System;

namespace ProyectoEjemplo
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

En la clase Program se crea una instancia de la clase Game1 y se invoca su método Run() que dará lugar al ciclo de ejecución del juego. Este ciclo ejecutará los métodos de la clase Game1 atendiendo al siguiente diagrama:

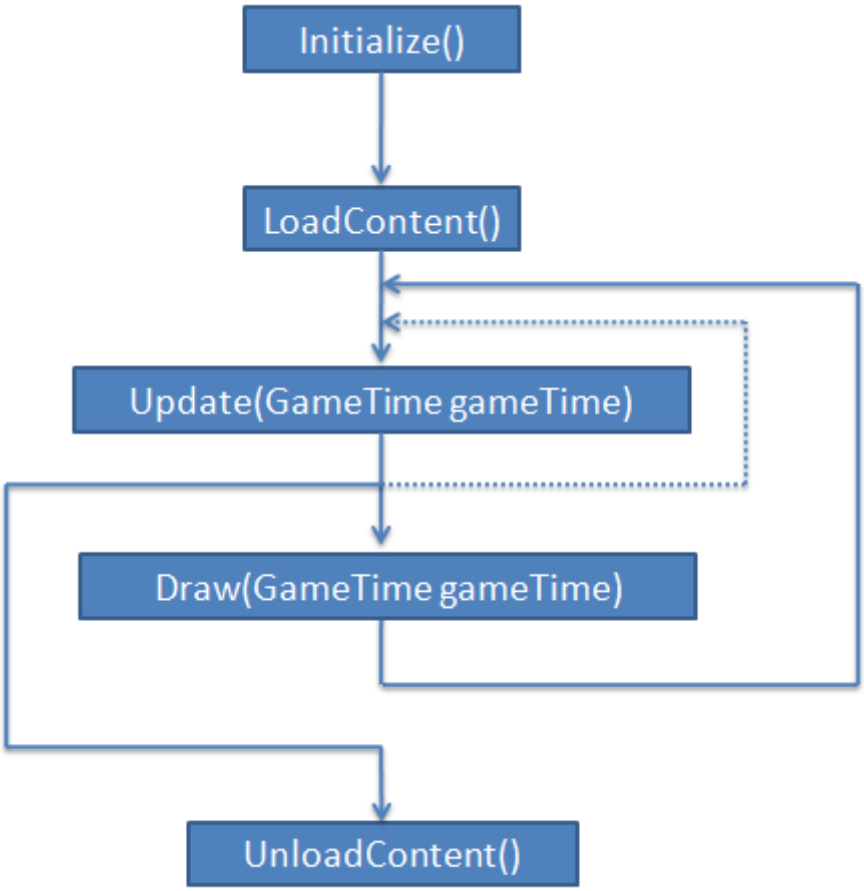

Imagen 33. Ciclo de ejecución de XNA

Una descripción más detallada de estos métodos es la siguiente:

- `protected override void Initialize()`

Este método será el primero del ciclo de ejecución y está pensado para inicializar toda la lógica. En concreto aquí se pueden iniciar timers, variables relacionadas con los FPS, físicas, etc. Es importante que todo este código esté antes de la instrucción "base.Initialize();", que es precisamente la encargada de hacer que todo lo que hay en la función Initialize() se lleve a cabo.

Finalizado este método, el ciclo de ejecución pasará a LoadContent().

- `protected override void LoadContent()`

En este método es donde se realizará la primera carga de contenido del juego como imágenes de fondo, sprites, modelos 3D, etc. Es importante que la carga se realice en este método y no antes ya que es aquí cuando ya se ha inicializado a través del método Initialize().

Finalizada la ejecución de este método, se pasa a Update(GameTime gameTime).

- `protected override void Update(GameTime gameTime)`

En este método se introducen los cambios en la lógica que se realizan a cada frame del juego. Por ejemplo, la interacción del usuario con el teclado y ratón, cálculo de coordenadas de jugadores, etc.

En el contenido del método Update no debería realizarse carga de gráficos, texturas o sonidos ya que supondría una caída en los FPS.

Como se muestra en el diagrama de forma punteada, el método Update(…) además de dar paso al método Draw(…) puede darse paso a sí mismo. Esto ocurre cuando el ordenador está procesando lentamente la lógica del método Update() por motivos externos (otros procesos), cuando se realizan cargas de contenido gráfico pesadas o hay muchos elementos en pantalla, cuando se está pausando la ejecución con el depurador…, en definitiva, cuando se penaliza la velocidad a la que se procesa el contenido del método, a lo que XNA responde con una caída de FPS buscando una actualización de la lógica más constante a costa de perder en dibujos por segundo.

Un último caso que se puede dar en este método es la ejecución de la instrucción “this.Exit()”, la cual dará paso a la ejecución del método UnloadContent().

- `protected override void Draw(GameTime gameTime)`

Esta es el método encargado del dibujado de gráficos en pantalla, se encargará de pintar los sprites, modelos 3D, textos y demás elementos gráficos.

Finalizado el proceso de dibujo, se volverá a ejecutar el método Update.

- `protected override void UnloadContent()`

En este método es donde se liberan los recursos cargados en memoria durante el juego antes de la finalización de éste.

4.3.4.Diseño de infraestructura MMO aplicado a XNA

En el caso de la aplicación cliente se tendrá que integrar el código dentro del ciclo de ejecución del juego en XNA, para lo cual se identifican los siguientes hitos en la comunicación:

1. Establecimiento de conexión con el Servidor de Conexión y recepción de datos del Servidor de Login y del Servidor de Juego. Si están todos los servidores activos se establecerá de conexión con el Servidor de Login
2. Envío de nombre de usuario y contraseña al Servidor de Login
3. Borrado, creación y selección de personajes con el Servidor de Login
4. Establecimiento de conexión con el Servidor de Juego
5. Intercambio de datos con el Servidor de Juego durante la partida

Definidos estos hitos, se propone el siguiente esqueleto como diagrama de clases para la implementación de la infraestructura MMO:

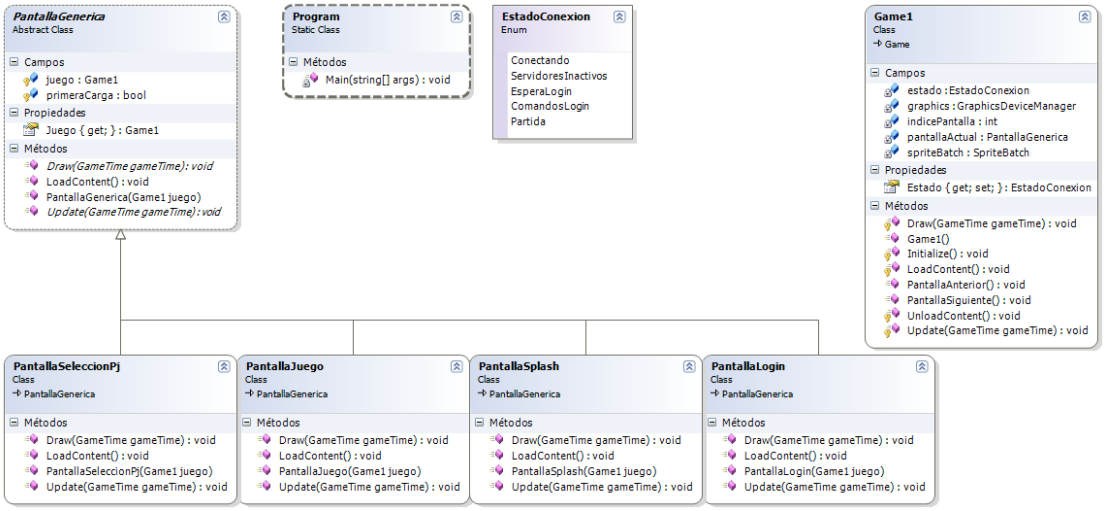

Imagen 34. Diseño de clases propuesto para aplicación MMO en XNA

La descripción de clases, atributos y métodos es la siguiente:

Enumerado	EstadoConexion
Descripción	Enumerado que identificará el estado en el que se encuentra el jugador dentro de los hitos definidos
Valor	Descripción
Conectando	El jugador ha arrancado el juego y está conectando con el Servidor de Conexión y esperando a que éste le indique la información del Servidor de Login y Servidor de Juego. Se aplica al hito 1 de comunicación.
ServidoresInactivo	El Servidor de Conexión responde que alguno de los servidores no está activo, o el propio Servidor de Conexión no es accesible. Se aplica al hito 2 de comunicación.
EsperaLogin	El servidor de Conexión ha respondido con la información de los servidores de Login y Juego, el jugador se ha conectado con el Servidor de Login y éste está esperando la autenticación por parte del jugador. Se aplica al hito 2 de comunicación.
ComandosLogin	El jugador se ha autenticado en el Servidor de Login, ahora puede enviar comandos de selección de personaje, borrar personajes existentes o crear nuevos. Se aplica al hito 3 de comunicación.
Partida	El jugador ha seleccionado un jugador en el Servidor de Login, se conecta con el Servidor de Juego y comienza la partida. Se aplica al hito 4 de comunicación.

Tabla 3. Enumerado EstadoConexion

Clase estática	Program
Descripción	Clase estática generada por XNA para comenzar la ejecución del Juego
Atributo	Descripción
Método	Descripción
main (string []args): static void	Método que sirve de punto de inicio para el juego.

Tabla 4. Clase estática Program

Clase	Game1
Descripción	Clase que dirige al resto de clases y guiará la ejecución del juego a través de sus fases.
Atributo	Descripción
- graphics: GraphicsDeviceManager	Gestor del dispositivo gráfico, permite establecer la resolución de pantalla, poner el juego a pantalla completa, etc.
- spriteBatch: SpriteBatch	Atributo para pintar en pantalla el contenido en 2 dimensiones y texto.
- pantallaActual: PantallaGenerica	Instancia de pantalla actual en la que se encuentra el jugador, a través de cada pantalla se avanzará por los distintos hitos.
- indicePantalla: int	Índice numérico de la pantalla en la que se encuentra el jugador.
- estado: EstadoConexion	Estado que identifica el hito de conexión en el que se encuentra el jugador.
Método	Descripción
+ Game1()	Constructor de la clase Game1, inicializa las variables de XNA.
+ Estado {get,set}:	Propiedad que permite obtener y establecer el estado actual en el

EstadoConexion	que se encuentra el jugador respecto a los hitos de conexión.
# Initialize(): void	Inicializa las variables introducidas en la clase Game1, estableciendo como PantallaActual una PantallaSplash, como indicePantalla el valor 0, y el estadoConexion lo establece a Conectando.
# LoadContent(): void	Se carga el contenido de la primera pantalla, en este caso, PantallaSplash.
# UnloadContent(): void	Libera los recursos del contenido cargado durante la partida.
# Update(GameTime gameTime): void	Llamará al método Update(GameTime gameTime) de la pantallaActual para actualizar la lógica en función de la pantalla.
# Draw(GameTime gameTime): void	Llamará al método Draw(GameTime gameTime) de la pantallaActual para dibujar los gráficos del juego.
+ pantallaSiguiente(): void	Al ser llamado este método, se cargará la siguiente pantalla en pantallaActual en función del índice almacenado por indicePantalla.
+ pantallaAnterior(): void	Al ser llamado este método, se cargará la pantalla anterior en pantallaActual en función del índice almacenado por indicePantalla.

Tabla 5. Clase Game1

Clase abstracta	*PantallaGenerica*
Descripción	Clase abstracta que define el comportamiento que tendrán las pantallas en el juego.
Atributo	Descripción
# juego: Game1	Instancia de la clase Game1 para acceder a los métodos de avanzar y retroceder pantalla.
# primerCarga: bool	Booleano que estando a TRUE indicará que aun no se ha cargado el contenido de la pantalla, por lo que en lugar de realizar Update(…) tendrá que realizar LoadContent().
Método	Descripción
# LoadContent(): void	Método abstracto en el que se cargará el contenido de la pantalla.
# Update(GameTime gameTime): void	Método en el que se actualizará la lógica de la pantalla.
# Draw(GameTime gameTime): void	Método en el que se dibujarán los gráficos.

Tabla 6. Clase abstracta PantallaGenerica

Clase	PantallaSplash: *PantallaGenerica*
Descripción	Pantalla que se muestra nada más ejecutar el juego. Pintará en pantalla el logotipo del juego, licencias, etc, y durante el tiempo que muestre esta información aprovechará para realizar el hito 1 de establecimiento de conexión con el Servidor de Conexión y recepción de datos de los servidores de Login y Juego. Una vez que se sabe si están disponibles los servidores o si hay error se establece la propiedad **juego.Estado** a su valor correspondiente: **EsperaLogin** en caso de que todo haya ido bien, o **ServidoresInactivos** en caso de error. A continuación, en el caso de que **juego.Estado** tenga el valor **EsperaLogin** se establecerá conexión con el Servidor de Login finalizando así el hito 1 de comunicación, y se establecerá **juego.Estado** a **EsperaLogin**. En última instancia se ejecuta el método **juego.pantallaSiguiente()**
Atributo	Descripción
juego: Game1	Atributo heredado

primerCarga: bool	Atributo heredado
Método	Descripción
LoadContent(): void	Métodos heredados
Update(GameTime gameTime): void	Métodos heredados
Draw(GameTime gameTime): void	Métodos heredados

Tabla 7. Clase PantallaSplash

Clase	PantallaLogin: *PantallaGenerica*
Descripción	Pantalla que se muestra después de PantallaSplash. En ella se mostrará una interfaz para realizar la autenticación en caso de que **juego.Estado** tenga el valor **EsperaLogin** o un mensaje de error en caso de que el **juego.Estado** sea **ServidoresInactivos**. Al cargar esta pantalla a través de la interfaz de autenticación se realizará el hito 2 de enviar el nombre de usuario y contraseña. Mientras los datos de autenticación no sean correctos se permanecerá en esta pantalla, y cuando una identificación sea correcta se pasará el **juego.Estado** a **ComandosLogin**, y se ejecutará el método **juego.PantallaSiguiente()**
Atributo	Descripción
juego: Game1	Atributo heredado
primerCarga: bool	Atributo heredado
Método	Descripción
LoadContent(): void	Métodos heredados
Update(GameTime gameTime): void	Métodos heredados
Draw(GameTime gameTime): void	Métodos heredados

Tabla 8. Clase PantallaLogin

Clase	PantallaSeleccionPj: *PantallaGenerica*
Descripción	Pantalla que se muestra después de PantallaLogin. En ella se mostrará una interfaz para realizar las acciones de selección, borrado y creación de personajes cubriéndose así las necesidades del hito 3 de comunicación. Una vez se realice la acción de selección de un personaje, se ejecutará el método **juego.PantallaSiguiente()**
Atributo	Descripción
juego: Game1	Atributo heredado
primerCarga: bool	Atributo heredado
Método	Descripción
LoadContent(): void	Métodos heredados
Update(GameTime gameTime): void	Métodos heredados
Draw(GameTime gameTime): void	Métodos heredados

Tabla 9. Clase PantallaSeleccionPj

Clase	PantallaJuego: *PantallaGenerica*
Descripción	Pantalla que se muestra después de PantallaSeleccionPj. En esta pantalla es donde se manejará la lógica y se pintarán los gráficos de la partida. En su carga de contenido, establecerá conexión con el Servidor de Juego y una vez empiece la comunicación de información en ambos sentidos se establecerá **juego.Estado** con el valor **Partida** realizando así el hito 4 de la comunicación.
Atributo	Descripción
juego: Game1	Atributo heredado
primerCarga: bool	Atributo heredado
Método	Descripción
LoadContent(): void	Métodos heredados
Update(GameTime gameTime): void	Métodos heredados
Draw(GameTime gameTime): void	Métodos heredados

Tabla 10. Clase PantallaGenerica

5.Caso práctico: Antiheroe

En los siguientes apartados se mostrará un caso práctico de implementación de la solución propuesta a lo largo de los subapartados de 4.2 Infraestructura MMO, solución del problema, abordando la problemática desde las perspectivas servidor y cliente.

En el apartado de descripción se hará una descripción del proyecto Antiheroe, se explicará el origen del proyecto y el objetivo final de éste.

En los siguientes apartados se tratará el desarrollo e implementación de la problemática estudiada en el apartado 4 Manual de desarrollo de infraestructura MMO, abarcando los distintos servidores propuestos así como el cliente desarrollado haciendo uso de la librería XNA.

5.1.Descripción

Antihéroe

"*En una obra de ficción, personaje que, aunque desempeña las funciones narrativas propias del héroe tradicional, difiere en su apariencia y valores*"[37].

"En otras palabras, un antihéroe es un protagonista que vive por la guía de su propia brújula moral, esforzándose para definir y construir sus propios valores opuestos a aquellos reconocidos por la sociedad en la que vive."[38]

En su inicio el proyecto Antiheroe surge como idea de desarrollo de un juego estilo *sandbox* basado en misiones en las que fuera el jugador quien decidiera qué y cómo realizarlas dotándole así de mayor libertad dentro de las posibilidades del juego.

Durante el estudio de las diferentes características que sería deseable que presentara y la experiencia jugable que iba a ofrecer a los usuarios finales surgió la idea de la colaboración entre distintos jugadores introduciendo así la necesidad de desarrollo del modo multijugador.

Una vez definido como objetivo la inclusión de un modo multijugador cooperativo, se opta por la plataforma de desarrollo propuesta por Microsoft[39]: el uso de la librería de desarrollo de XNA[40] sobre el lenguaje de programación C#.

Es durante el estudio y desarrollo de la parte multijugador cuando se define el objetivo final el cual ha guiado al proyecto hasta la fecha.

Partiendo de la necesidad de compartir la experiencia presentada por el juego con varios jugadores se planteó la pregunta ¿cuántos jugadores se podrían soportar en una misma partida? La solución a esta pregunta es sencilla, depende de la implementación.

De aquí surgió el nuevo objetivo final, identificar la infraestructura necesaria para dar soporte a jugadores de forma masiva.

Con este nuevo objetivo se presenta toda la problemática estudiada hasta este apartado y a la cual se dará una solución implementada en los siguientes subapartados.

5.2.Aplicación cliente

La aplicación cliente será aquella que ejecute el usuario final, es decir, los jugadores. Será desarrollada en lenguaje C# utilizando la librería de desarrollo XNA partiendo de la creación de un proyecto explicada en el apartado 4.3.2. Pasos para crear un proyecto de juego bajo XNA.

Será la encargada de presentar la interfaz gráfica al jugador para que interactúe con el sistema, enviando esta información a los distintos servidores y recibiendo a su vez la información de los servidores para presentar el estado actual de la partida validado por éstos.

A continuación se explica la codificación que da solución a la propuesta de los distintos subapartados de 4.2 Infraestructura MMO, solución del problema.

5.2.1.Cámara

Un problema de obligado tratado en el desarrollo de cualquier juego es el sistema de cámara que se va a utilizar.

En este caso se optará por una perspectiva aérea en 2 dimensiones, de modo que los jugadores y mapeado se verán como si la cámara les enfocara desde una perspectiva lateral pero elevada sobre unos 90º del eje del suelo, siendo ejemplos de esta cámara juegos como Pokemon o Zelda.

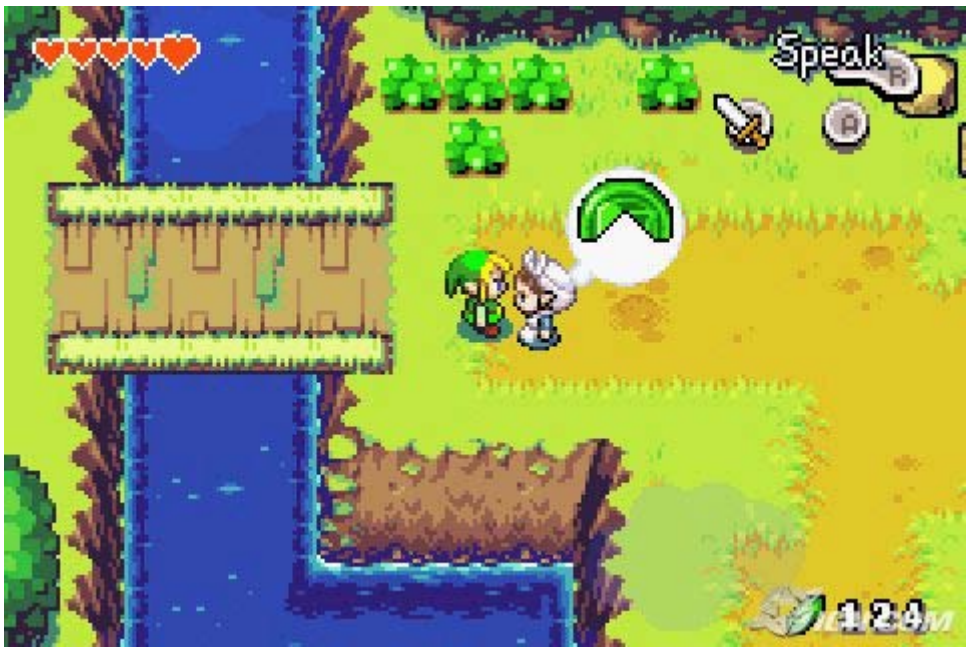

Imagen 35. Perspectiva aérea de Zelda – The Minish Cap

Dado que la perspectiva aérea será sobre 2D, el trabajo relacionado con la creación de una sensación de profundidad en lugar de ser competencia de la lógica de la propia programación como sería en el caso del diseño 3D, recaerá sobre el apartado de diseño

gráfico, de modo que los modelos a diseñar tendrán que estar representados como si la cámara los enfocara desde el ángulo deseado.

Por otro lado será necesario establecer un sistema de coordenadas en el que basar la ubicación del jugador respecto al mapa, teniendo distintas posibilidades como son: tener un mapa estático por el que el jugador se moverá y pudiendo pasar de una pantalla a otra alcanzando los extremos de la pantalla u objetos como puertas, escaleras, etc; tener al jugador fijado en el centro de la pantalla de forma estática y que sea el mapa lo que se desplaza en función de su movimiento; y una última opción en la que se mezclarían ambas, manteniendo al jugador en el centro de la pantalla y moviendo el mapa salvo cuando se aproxima a los extremos de éste, momento en el que pasará a desplazarse el jugador sobre el mapa.

En este caso se propone una implementación en la que el jugador queda ubicado de forma estática en el centro de la pantalla y lo que se desplaza es el mapa.
Para ello, será necesario contar con dos sistemas de coordenadas: unas coordenadas absolutas en las que la posición (X,Y) = (0,0) serán la esquina superior izquierda del mapa y servirán para saber dónde se encuentra el jugador en relación con el mapa; y unas coordenadas relativas que servirán para pintar en pantalla los gráficos y en las que el jugador estará fijado de forma estática en el centro de la pantalla delegando de este modo las coordenadas (X,Y) del jugador a la resolución establecida por el tamaño de la pantalla de juego.

De este modo, si se tuviera una resolución de pantalla de 800x600, la coordenada relativa que definiría donde está ubicado el jugador, al ser el centro de la pantalla y estableciendo el crecimiento de las coordenadas del eje X hacia la derecha y de las coordenadas del eje Y hacia abajo, sería la coordenada C (400,300) según se muestra en la siguiente imagen:

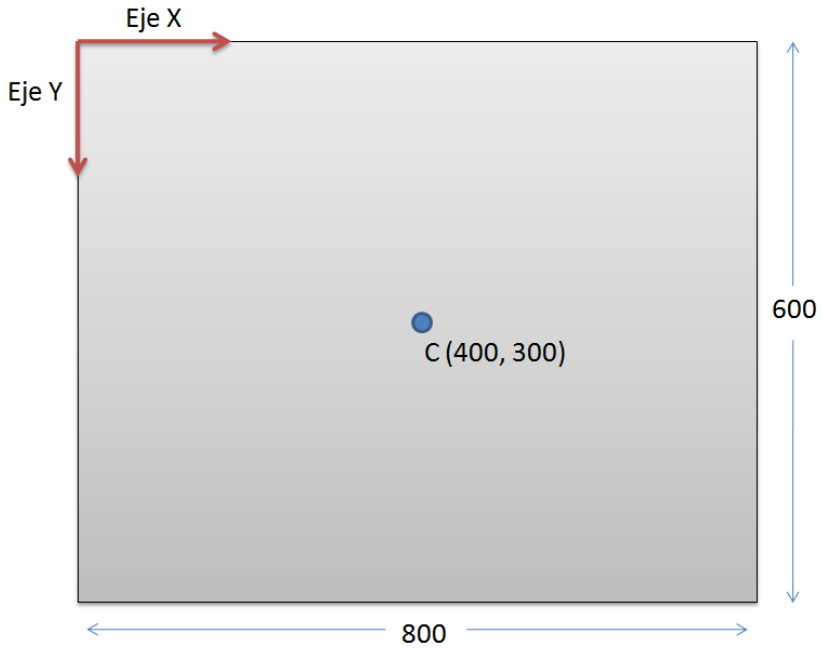

Imagen 36. Ejes de coordenadas de representación de gráficos

Aplicando esta idea, si el jugador se encontrara en la coordenada (0,0) de un mapa representado sobre una resolución de 800x600, los valores de este sistema de coordenada serían:

- Coordenada absoluta del jugador: (X,Y) = (0,0)
- Coordenada relativa del jugador: (X,Y) = (400,300)

Y de forma gráfica se tendría al jugador en el centro de la pantalla y sobre la esquina superior izquierda del mapa como muestra el siguiente gráfico:

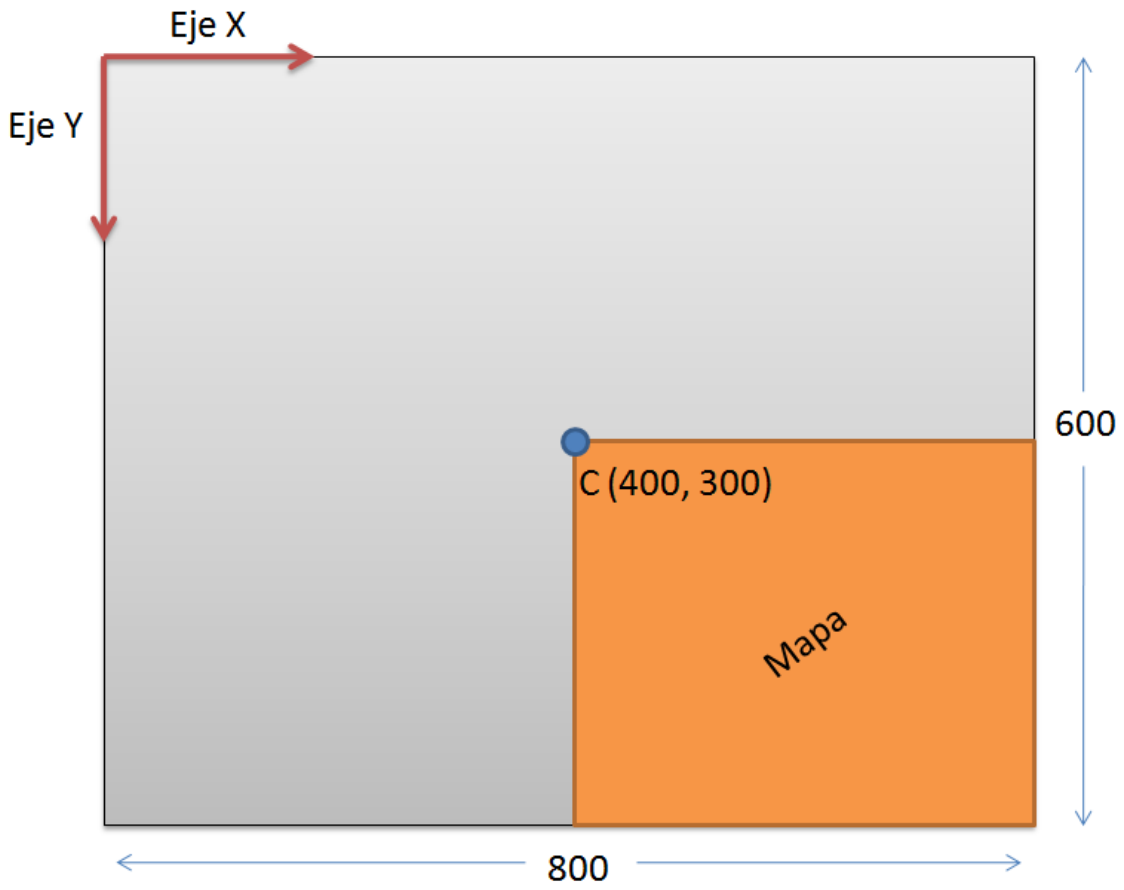

Imagen 37. Representación de coordenadas - jugador sobre el mapeado

Basado en este sistema de coordenadas, se podría controlar el movimiento y ubicación de los objetos a través de la diferencia de la siguiente ecuación:

$$Rx = Ax – Ac + Rc$$

Donde Rx será la posición relativa del objeto y por tanto la coordenada donde se pintará dicho objeto, Ax será la posición absoluta del objeto y por tanto la posición del objeto respecto a la esquina superior izquierda del mapa, Rc será la posición relativa del jugador y por tanto el punto central de la pantalla que en el caso de una resolución de 800x600 será Rc(400,300), y finalmente Ac será la posición absoluta del jugador que en este caso al estar en la esquina superior izquierda del mapa será Ac(0,0).

De este modo si sabemos:

- Coordenada absoluta del jugador: Ac (0,0)
- Coordenada absoluta del objeto (en este caso el mapa): Ax (0,0)
- Coordenada relativa del jugador: Rc (400,300)

Podemos calcular la coordenada donde se dibujará el objeto:

- Coordenada relativa del mapa: Rx = Ax – Ac +Rc = (0,0) – (0,0) + (400,300) = (400,300)

A continuación se muestra otro ejemplo teórico del funcionamiento de este sistema dual de coordenadas:

El jugador se encuentra en la esquina superior izquierda del mapa como se vio en la Imagen 37, por lo que se tiene:

- Coordenada absoluta del jugador: Ac = (0,0)
- Coordenada absoluta del mapa: Ax = (0,0)
- Coordenada relativa del jugador: Rc = (400,300)
- Coordenada relativa del mapa: Rx = Ax – Ac +Rc = (0,0) – (0,0) + (400,300) = (400,300)

Si en esta situación el jugador se desplaza 100 posiciones hacia abajo sobre el eje Y, es decir, el jugador se encuentra 100 posiciones debajo de la esquina superior del mapa, se tendría la siguiente situación:

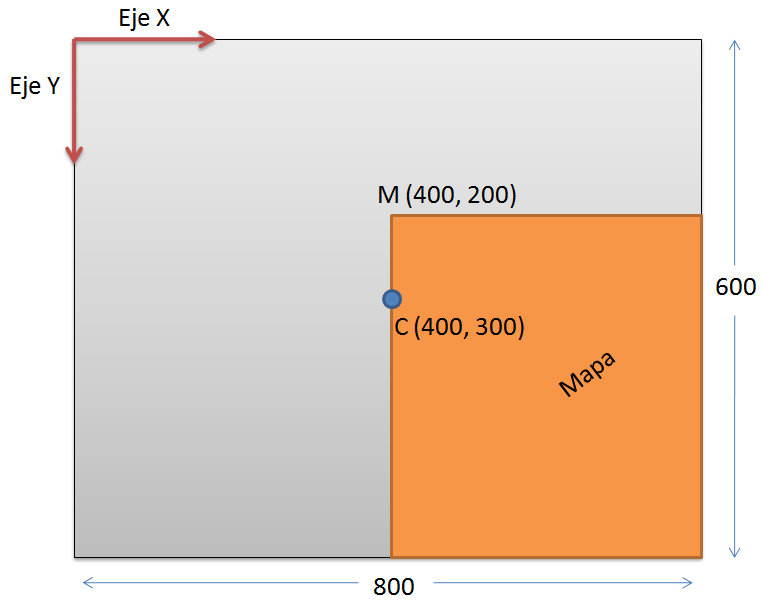

Imagen 38. Sistema de coordenadas. Desplazamiento del jugador sobre el mapeado

De la Imagen 39 se tiene la siguiente información:

- Coordenada absoluta del jugador: Ac = (0,100)
- Coordenada absoluta del mapa: Ax = (0,0)
- Coordenada relativa del jugador: Rc = (400,300)
- Coordenada relativa del mapa: Rx=Ax-Ac+Rc = (0,0)-(0,100)+(400,300)=(400,200)

De igual modo, si en esta situación se quisiera dibujar un objeto de color verde encontrado en la posición absoluta (100,100) y por tanto desplazado 100 posiciones a la derecha y 100 posiciones hacia debajo de la esquina superior izquierda del mapa, se tendría:

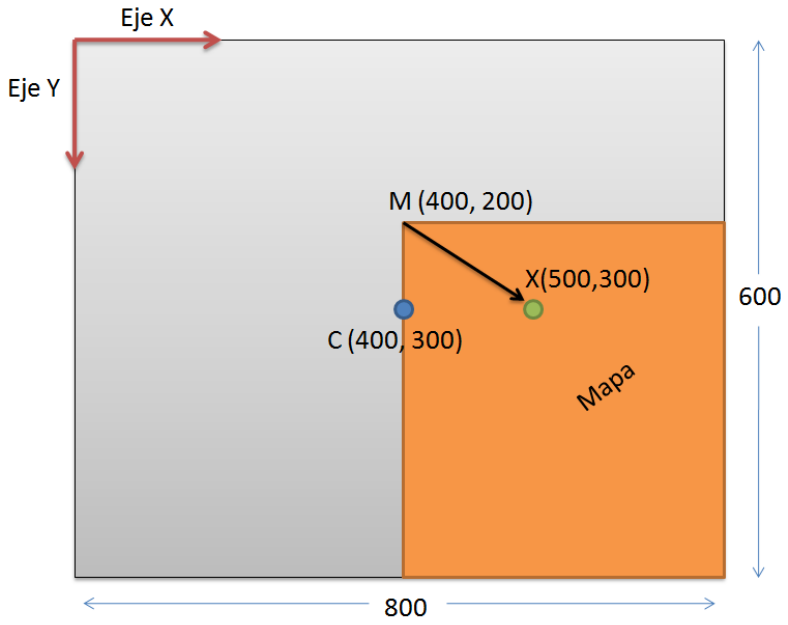

Imagen 39. Sistema de coordenadas. Elementos distintos del jugador en el mapeado

- Coordenada absoluta del jugador: Ac = (0,100)
- Coordenada absoluta del objeto verde: Ax = (100,100)
- Coordenada relativa del jugador: Rc = (400,300)
- Coordenada relativa del objeto: Rx=Ax-Ac+Rc = (100,100)-(0,100)+(400,300)=(500,300)

Basado en estos cálculos, se implementa en una clase Cámara esta funcionalidad a través del siguiente código:

```
//Código para el cálculo de los fragmentos de mapa
public Vector2 CalcularPosicionMapa(ImagenTileSet imagen)
{
      Vector2 Ac = _pantalla.JugadorLocal.PosicionMovimiento;
      Vector2 Ax = imagen.PosicionOriginal;
      Vector2 Rc = new Vector2(_pantalla.Juego.Width * 0.5f,
      _pantalla.Juego.Height * 0.5f);
      return (Ax - Ac + Rc);
}

//Código para el cálculo de posiciones de jugadores
public Vector2 CalcularPosicionJugador(Jugador j)
{
      if (j == _pantalla.JugadorLocal)
```

```
        {
              return new Vector2(_pantalla.Juego.Width * 0.5f,
              _pantalla.Juego.Height * 0.5f);
        }
        Vector2 Ax = j.PosicionMovimiento;
        Vector2 Ac = _pantalla.JugadorLocal.PosicionMovimiento;
        Vector2 Rc = new Vector2(_pantalla.Juego.Width * 0.5f,
        _pantalla.Juego.Height * 0.5f);
        return (Ax - Ac + Rc);
}
```

Definido el código para calcular las posiciones en las que será pintado el mapa y los jugadores, sólo queda establecer que éstas serán calculadas al final del método Update que será llamada dentro del ciclo de ejecución seguido por XNA y que quedó explicado en 4.3.3. Resumen descriptivo del ciclo de ejecución de un juego en XNA:

```
public override void Update()
{
        //Código para el cálculo de posiciones de jugadores y mapas
        //Se ubica a cada jugador en su posición
        for (int i = 0; i < _jugadores.Count; i++)
        {
              _jugadores[i].Grafico.Update();
              _jugadores[i].Grafico.Posicion =
              _camara.CalcularPosicionJugador(_jugadores[i]);
              _jugadores[i].Grafico.SituarCentro(_jugadores[i].Grafico.Po
        sicion);
        }
        //Se calculan las coordenadas de posición del mapa
        _mapa.Update();
}
```

En el código presentado se hace uso de dos funciones que no han sido explicadas, éstas son:

- **jugadores[i].Grafico.SituarCentro(_jugadores[i].Grafico.Posicion);**

  Con el fin de estandarizar el cálculo de las coordenadas de todos los objetos que se puedan representar en el mapa, y dado que se podría dar el caso de que no todos tuvieran las mismas dimensiones, cuando se calculan las coordenadas de posición de un objeto a representar en la pantalla estas hacen referencia a la esquina inferior derecha del gráfico y por tanto es necesario añadir un cálculo adicional que será el que ubique el centro del gráfico en la coordenada calculada, este cálculo consistirá en restar a la coordenada de la esquina inferior derecha del gráfico la mitad de su ancho y la mitad de su alto:

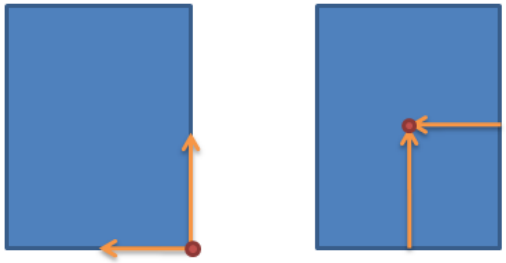

Imagen 40. Calculo del centro de un objeto a representar gráficamente

Por lo que el código será:

```
public void SituarCentro(Vector2 posicion)
{
      int ancho = _imagenFondo.ObtenerAncho();
      int alto = _imagenFondo.ObtenerAlto();
      _posicion = new Vector2(
             (float)(posicion.X - (ancho * 0.5f)),
             (float)(posicion.Y - (alto * 0.5f))
            );
}
```

- **_mapa.Update();**

Como se explicará en el punto 5.2.3 Carga de contenido, el mapa se divide en fragmentos, por ello a través de una clase que representa al mapa cargado en pantalla se realizará la llamada a la función apropiada para el cálculo de coordenadas de cada porción de mapa:

```
public class Mapa
{

      . . .

      public void Update()
      {
            for (int i = 0; i < dim_fila; i++)
            {
                  for (int j = 0; j < dim_columna; j++)
                  {
                        tiles[i,j].Posicion =
                        pantalla.Camara.CalcularPosicionMapa(
                              tiles[i, j]
                              );
                  }
            }
      }

}
```

5.2.2.Movimiento del jugador

Controlar el movimiento del jugador consta de dos fases:

1. Capturar la interacción del usuario con el teclado/ratón
2. Actualizar las coordenadas del personaje en base a la captura de la fase 1 para que la cámara que se definida en el apartado anterior pueda representar el estado actual de la partida.

Para realizar la primera fase se definirán en la clase principal Juego los siguientes atributos:

```
//Para controlar la entrada del usuario
private KeyboardState miTecladoActual;
private KeyboardState miTecladoAnterior;
private MouseState miRatonActual;
private MouseState miRatonAnterior;
private Controles _controles;
```

En estos atributos se guardará el estado del instante anterior y actual del teclado y ratón, además de una instancia de la clase Controles que define que teclas están asociadas a los movimientos (arriba, abajo, izquierda, derecha, etc.)

La necesidad de guardar el estado anterior de las posiciones surge para saber cuándo se ha presionado una tecla, cuando se está manteniendo presionada y cuando se suelta por ejemplo para el sistema de chat, ya que si no tuviéramos un estado anterior con el que contrastar el estado actual, al presionar una determinada tecla, se escribiría tantas veces como veces se llame al método Update() por segundo.

Esta captura de la interacción del usuario se realizará al comienzo del método Update() para que a continuación se actualice toda la lógica del juego y se pueda representar a través del método Draw() el estado actual de la partida.

El código es el siguiente:

```
protected override void Update(GameTime gameTime)
{
      // Allows the game to exit
      if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
      ButtonState.Pressed)
            this.Exit();

      // TODO: Add your update logic here
      this.CapturarEntrada();
}

private void CapturarEntrada()
{
      miTecladoAnterior = miTecladoActual;
      miTecladoActual = Keyboard.GetState();
      miRatonAnterior = miRatonActual;
      miRatonActual = Mouse.GetState();
}
```

Una vez se tiene capturada la interacción del usuario se pasa a la fase 2 en la que se actualizan las coordenadas del jugador en función del movimiento del jugador, comprobando si hay una diferencia de estado frente al instante de partida anterior:

```
private int DetectarMovimiento()
{
	Controles controles = _juego.Controles;
	if (_juego.TecladoActual.IsKeyDown(controles.Arriba))
		return Cliente.ARRIBA;
	else if (_juego.TecladoActual.IsKeyDown(controles.Abajo))
		return Cliente.ABAJO;
	else if (_juego.TecladoActual.IsKeyDown(controles.Izquierda))
		return Cliente.IZQUIERDA;
	else if (_juego.TecladoActual.IsKeyDown(controles.Derecha))
		return Cliente.DERECHA;
	//Si no se está presionando ninguna de esas teclas se está
	quieto
	return Cliente.QUIETO;
}


public override void Update()
{
	bool nuevasTeclas = false;
	bool colision = false;

	if (_juego.TecladoActual != _juego.TecladoAnterior)
		nuevasTeclas = true;
	int mov = this.DetectarMovimiento();

	Vector2 posicion = _jugadorLocal.PosicionMovimiento;
	_jugadorLocal.FinalizarMovimiento(mov, posicion);

	//Código para el cálculo de posiciones de jugadores y mapas
	. . .

}
```

En el caso de la llamada al método **_jugadorLocal.FinalizarMovimiento(mov, posicion);**, el método se encuentra en la clase que define el comportamiento del jugador, y contiene el siguiente código:

```
public void FinalizarMovimiento(int nuevoMovimiento, Vector2 posicion)
{
	this.movimiento = nuevoMovimiento;
	this.instante = DateTime.Now;
	this.posicionAbsoluta = new Vector2(posicion.X, posicion.Y);
}
```

5.2.3.Carga de contenido gráfico

El contenido que se cargará para representar en pantalla constará de:

- Imágenes estáticas que serán utilizadas como fondos de pantalla o complementos para los elementos de la interfaz con la que interactuará el usuario
- Imágenes definidas en tilesets que serán utilizadas para generar el mapa del juego, o para facilitar la animación de los personajes
- Textos que serán utilizados para comunicar información al jugador desde la pantalla de juego.

Con el fin de utilizar un formato de imágenes que ofrezca una buena relación en cuanto a funcionalidad añadida, compresión de imagen y calidad de ésta, se optará por el formato PNG el cual permitirá jugar con una escala de colores suficientes, comprimir los datos de la imagen y como añadido permite usar un canal alfa para definir la transparencia de pixels permitiendo mostrar imágenes que se encuentren debajo de una transparente.

- **Carga de imágenes estáticas**

Las imágenes estáticas serán imágenes que durante la ejecución del juego no varíen ni en su diseño ni en su posición, como por ejemplo la imagen de introducción del juego:

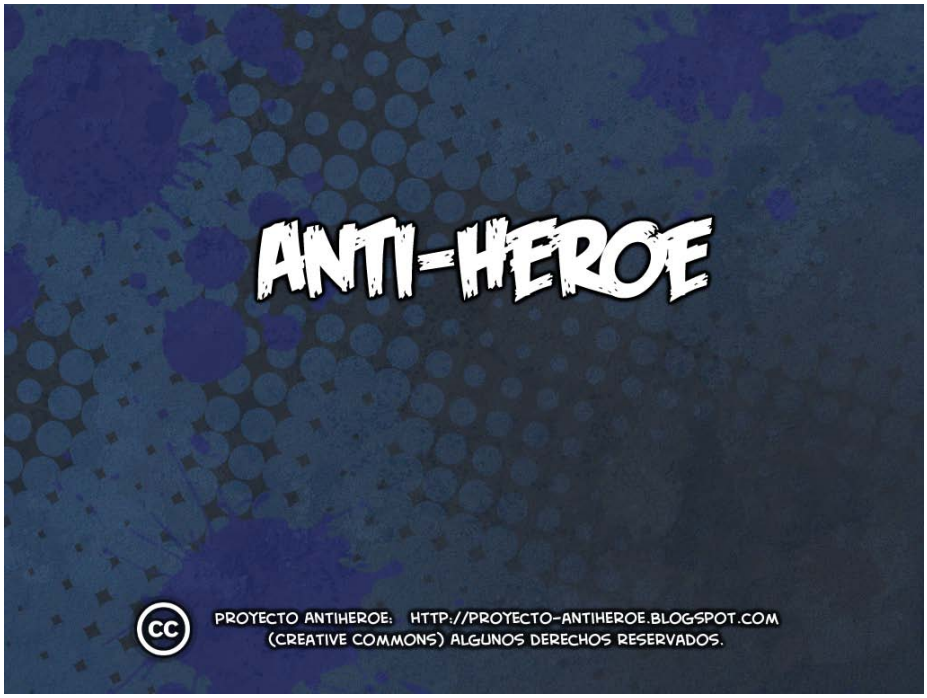

Imagen 41. Captura de pantalla de PantallaSplash de AntiHeroe en ejecución

Para realizar la carga de las imágenes estáticas se tiene una clase Imagen con los siguientes atributos y constructor:

```
// Textura de la imagen
private Texture2D _textura;
// Indica si se ha redimensionado la textura
private bool _redimensionar;
// Ancho de la imagen
private int _ancho;
// Alto de la imagen
private int _alto;

public Imagen(ContenedorGenerico contenedor, Vector2 posicion, string
id, string textura): base(contenedor, posicion, id)
{
      _textura =
      _contenedor.Pantalla.Juego.Content.Load<Texture2D>("./Imagenes/"
      + textura);
      _tileX = -1;
      _tileY = -1;
}
```

De modo que cuando se quiera instanciar una imagen se utilizará el siguiente código:

```
Imagen _imagenFondo = new Imagen(this, new Vector2(),
"fondo_de_contenedor", textura, (int)tam.X, (int)tam.Y);
```

- **Carga de imágenes definidas en tilesets**

En la implementación del juego se darán dos casos de imagen definida en tileset: los personajes de los jugadores para producir la animación de éstos; y el mapa de juego para tener en un mismo gráfico todos los elementos que se puedan dibujar.

Para el caso del tileset que contiene al personaje del jugador se necesitará a nivel gráfico un tileset dividido en celdas de tamaño 50x50 pixels.

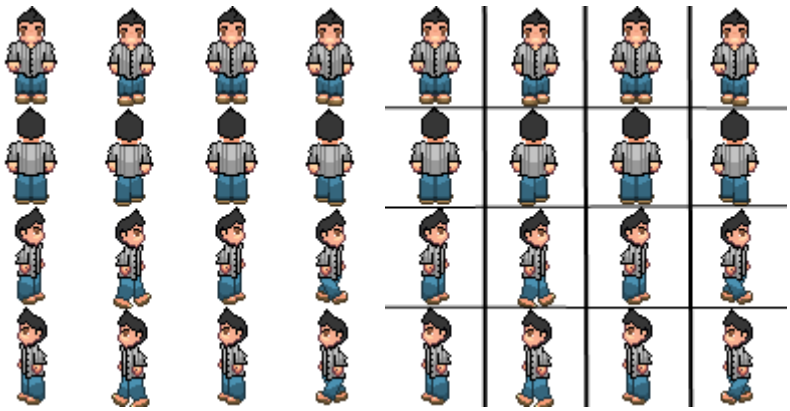

Imagen 42. Tileset del personaje principal

Para representarlo se utilizará la clase Imagen antes definida, sólo que al quererse representar sólo un fragmento de la imagen y no su totalidad, se harán uso de los siguientes atributos antes no introducidos:

```
//Desplazamiento en coordenadas X de la celda a dibujar
private int _tileX;
//Desplazamiento en coordenadas Y de la celda a dibujar
private int _tileY;
//Tamaño de cada celda en el eje X
private int _tamTileX;
//Tamaño de cada celda en el eje Y
private int _tamTileY;
//Frame actual para dibujar
private int _frameActual = -1;
//Máximo número de frames por fila en el tileset
private const int _frameMax = 4;
//Velocidad a la que se produce la animación
private const int _velocidadAnimacion = 12;
//Movimiento realizado anteriormente
private int _movimientoAnterior;
```

El constructor de este tipo de imágenes será el siguiente:

```
public Imagen(ContenedorGenerico contenedor, Vector2 posicion, string
id, string textura, int tileX, int tileY, int tamTileX, int tamTileY)
    : base(contenedor, posicion, id)
{
    _textura =
    _contenedor.Pantalla.Juego.Content.Load<Texture2D>("./Imagenes/" +
    textura);
    _redimensionar = false;
    _tileX = tileX;
    _tileY = tileY;
    _tamTileX = tamTileX;
    _tamTileY = tamTileY;
    _movimientoAnterior = Cliente.ABAJO;
}
```

Que será llamado del siguiente modo:

```
Imagen    _personaje    =    new    Imagen(this,    new    Vector2(60,
_seleccionar.Posicion.Y + 10), "imagen_" + nombre, textura, 0, 0, 50,
50);
```

Para el caso de la carga del mapa de juego, al ser otro tileset, se cargará la textura igual que se hizo en la representación de personajes y además será necesario contar con dos elementos:

- Un tileset dividido en celdas de tamaño 50x50 pixels con los gráficos para poder representar el mapeado.

Imagen 43. Tileset del mapa de juego (sin división y con división en celdas)

- Un fichero de texto que define de qué forma se dispondrán las celdas del mapeado y que durante su carga permitirá definir qué elementos no se podrán atravesar, como se verá en el apartado 5.2.7. Sistema de colisiones:

```
#Textura
mapaRaste
Dimensiones (Y,X)
20 20
#Matriz de gráficos
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 37 38 17 17 17 17 25 26 17 17 17 17 10 10
10 10 17 17 17 11 47 48 11 11 11 11 35 36 17 17 17 17 10 10
10 10 17 21 22 22 22 22 22 22 22 22 22 23  9 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

De este modo si se asocia a cada celda del tileset un número, partiendo de 0 en la esquina superior izquierda y creciendo de izquierda a derecha y de arriba a abajo, y pintamos en pantalla la matriz del fichero de texto en base a los números del tileset, tendríamos:

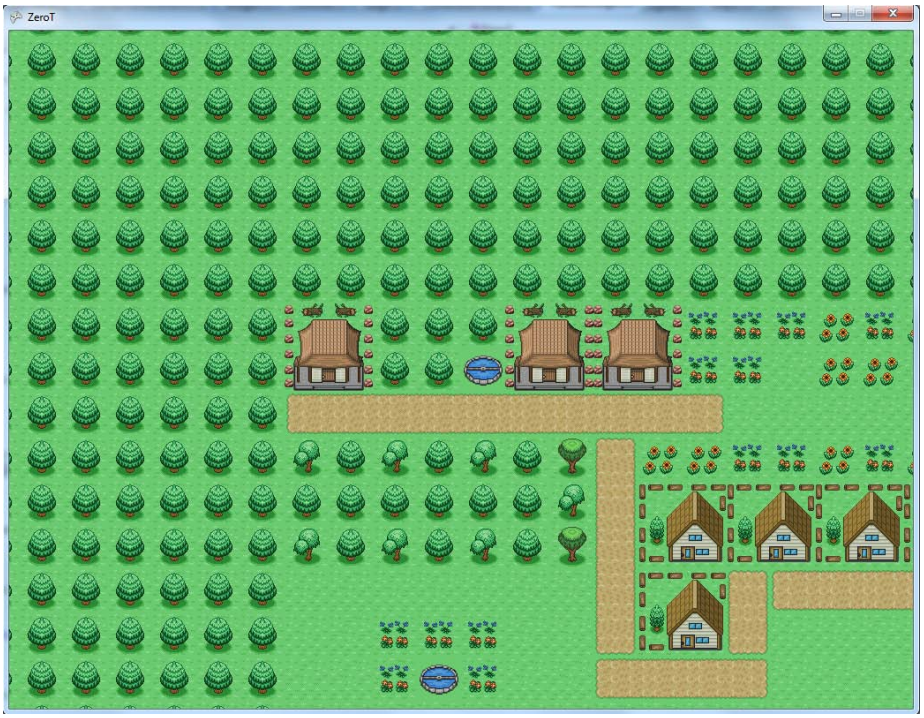

Imagen 44. Resultado de cruzar tileset de mapeado con el fichero de descripción del mapa

El código que se encarga de unificar el tileset del mapeado con la matriz numérica definida en el fichero de texto se encontrará en una clase Mapa que define el comportamiento de éste. Para ello contará con los siguientes atributos:

```
//Nombre del fichero con el contenido del mapa
private String fichero;
//Textura del sprite
private String textura;
//Pantalla en la que se encuentra el mapa
private PantallaJuego pantalla;
//Identificador del sprite
private String id;
//Tamaño de la primera dimensión
private int dim_fila;
//Tamaño de la segunda dimensión
private int dim_columna;
//Logica leída del fichero. Asigna un número a cada tile de la textura
private int[,] logica;
//Matriz con los gráficos para dibujar el mapa. Se crea a partir de la
lógica.
private ImagenTileSet[,] tiles;
```

Para instanciar un objeto de este tipo se introducirá el siguiente código:

```
_mapa = new Mapa("mapa.txt", "mapeado", this);
```

Y el código encargado de cargar el mapa en el juego será el siguiente:

```
public Mapa(String fichero, String id, PantallaJuego pantalla)
{
    this.fichero = fichero;
    this.id = id;
    this.pantalla = pantalla;

    //Procesamos el fichero
    StreamReader lector;
    String linea = "";
    int contador = 0;
    char[] delim = new char[1];
    delim[0] = ' ';
    int tam_tile = pantalla.Juego.TamanoTile;
    String[] cadenas;

    mapa = new ContenedorGenerico(pantalla, new Vector2(), null,
"mapa");

    Imagen fondo = new Imagen(mapa, new Vector2(), "", "map_prueba");
    mapa.AgregarElemento(fondo);

    try
    {
        lector = File.OpenText("./mapas/" + fichero);

        // Lee linea a linea
        while ((linea = lector.ReadLine()) != null)
        {
            Vector2 posicion = new Vector2();
            if (!linea.Contains("#")) //En caso de que no sea un
comentario
            {
                switch (contador)
                {
                    case 0:
                        {//Se está leyendo la textura del mapa
                            this.textura = linea;
                            break;
                        }
                    case 1:
                        {//Se está leyendo las dimensiones del mapa
                            cadenas = linea.Split(delim);
                            dim_fila = Int32.Parse(cadenas[0]);
                            dim_columna = Int32.Parse(cadenas[1]);
                            tiles = new ImagenTileSet[dim_fila,
dim_columna];
                            logica = new int[dim_fila, dim_columna];
                            break;
                        }
                    default:
                        {//Se está leyendo los gráficos del mapa
                            cadenas = linea.Split(delim,
StringSplitOptions.RemoveEmptyEntries);

                            posicion = new Vector2(0, (contador - 2) *
50);

                            int aux;
                            for (int i = 0; i < cadenas.Length; i++)
```

```
                            {
                                aux = Int32.Parse(cadenas[i]);

                                tiles[contador - 2, i] = new
ImagenTileSet(textura, posicion, aux, pantalla, (contador-2)+" "+i);

                                posicion += new Vector2(50, 0);

                                if ((aux == 10) || (aux == 37) || (aux
== 38) || (aux == 47) || (aux == 48) ||
                                    (aux == 25) || (aux == 26) || (aux
== 35) || (aux == 36) || (aux == 8) ||
                                    (aux == 11) || (aux == 12) || (aux
== 13))
                                {
                                    logica[contador - 2, i] = 1; //No
se puede pisar
                                }
                                else
                                {
                                    logica[contador - 2, i] = 0; //Se
puede pisar
                                }
                            }

                            posicion += new Vector2(0, 50);

                            break;
                        }
                    }
                    contador++;
                }
            }
            // Cerramos la conexion
            lector.Close();
        }
}
```

Como se puede observar, se accede a la ruta del mapa, se abre el fichero y se va leyendo línea a línea donde:

- Si encuentra una línea que comienza con **#** es un comentario
- En la primera línea obtiene el nombre de la textura con el tileset del mapa
- En la segunda tiene las dimensiones del mapa para la creación de las variables necesarias que van a contener el mapa
- En las sucesivas líneas obtiene la celda del tileset de donde cargará el fragmento de imagen del mapa guardando esta información en la matriz **tiles[]** y matriz **logica[]** si se puede pisar o no para el control posterior de colisiones

- **Carga de texto**

Finalmente la carga de contenido textual se realizará a través de una clase Texto definida para representar el comportamiento de éste y que constará de los siguientes atributos:

```
// Estilo utilizado en el texto
private string _estilo;
// Texto que se mostrará
private string _texto;
// Color del texto
private Color _color;
// Origen de coordenadas del texto (izquierda o centrado)
private string _origen;
```

Su constructor recibirá los siguientes parámetros:

```
public Texto(ContenedorGenerico contenedor, Vector2 posicion, string
id, string estilo, string texto, string origen, Color color)
: base(contenedor, posicion, id)
{
	_estilo = "./Fuentes/" + estilo;
	_texto = texto;
	_color = color;
	_origen = origen;
}
```

Y será instanciado del siguiente modo:

```
Texto tUsuario = new Texto(res, new Vector2(), "texto_usuario",
"Defecto", "nombre de usuario:", "izquierda", Color.White);
```

Al introducir la posibilidad de utilizar distintos formatos será necesario crear un objeto de tipo SpriteFont por cada estilo y posteriormente cargarlos con el siguiente código:

```
SpriteFont sf = Content.Load<SpriteFont>(_estilo);
```

5.2.4.Representación en pantalla y animación de los personajes

Como se explicó en el apartado anterior, para realizar la animación de los personajes se va a recurrir al uso de sprites o tilesets, dado que el juego a implementar será en 2 dimensiones.

Con el fin de controlar el comportamiento de estos gráficos se tendrá una clase Imagen que guardará la textura cargada, el ancho, alto del gráfico, y la coordenada en que se dibujará en la pantalla, y otra clase ImagenTileSet que servirá para controlar los tiles del mapeado.

Siguiendo la división establecida en el apartado anterior, se tienen tres tipos de contenido:

- **Imágenes estáticas**

En el caso de las imágenes estáticas no se producirá animación, por lo que para dibujar la imagen se recurrirá a la siguiente línea de código en todas las iteraciones del método Draw(), donde la variable **j** será la instancia de la clase principal Juego:

```
j.SpriteBatch.Draw(_textura, _posicion, Color.White);
```

- **Imágenes definidas en tilesets**

En los gráficos creados como tilesets se vieron dos posibilidades, representar un jugador y un mapa.

En el caso del jugador se utilizará la clase Imagen la cual contará con los siguientes atributos:

```
//Desplazamiento en coordenadas X de la celda a dibujar
private int _tileX;
//Desplazamiento en coordenadas Y de la celda a dibujar
private int _tileY;
//Tamaño de cada celda en el eje X
private int _tamTileX;
//Tamaño de cada celda en el eje Y
private int _tamTileY;
//Frame actual para dibujar
private int _frameActual = -1;
//Máximo número de frames por fila en el tileset
private const int _frameMax = 4;
//Velocidad a la que se produce la animación
private const int _velocidadAnimacion = 12;
//Movimiento realizado anteriormente
private int _movimientoAnterior;
```

Finalmente, para realizar la animación utilizará un método de dibujo que recibirá el movimiento que se está realizando:

```
public void Draw(int movimiento)
{
	if (!_mostrar)
		return;

	Juego j = _contenedor.Pantalla.Juego;

	if (movimiento != Cliente.QUIETO)
	{
		_movimientoAnterior = movimiento;
	}

	if (_frameActual >= (_frameMax * _velocidadAnimacion))
		_frameActual = 0;
	int img = _frameActual /_velocidadAnimacion;
	if (movimiento == Cliente.QUIETO)
	{
	//Si el movimiento actual es quieto, dejamos al personaje
	mirando hacia el último movimiento
		if (_movimientoAnterior == Cliente.ABAJO)
		{
			j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
			_posicion, new Rectangle(0, 0, _tamTileX, _tamTileY),
			Color.White);
		}
		else if (_movimientoAnterior == Cliente.ARRIBA)
		{
			j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
			_posicion, new Rectangle(0, 50, _tamTileX,
			_tamTileY), Color.White);
		}
		else if (_movimientoAnterior == Cliente.DERECHA)
		{
			j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
			_posicion, new Rectangle(0, 100, _tamTileX,
			_tamTileY), Color.White);
		}
		else if (_movimientoAnterior == Cliente.IZQUIERDA)
		{
			j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
			_posicion, new Rectangle(0, 150, _tamTileX,
			_tamTileY), Color.White);
		}
	}
	else if (movimiento == Cliente.ABAJO)
	{
		j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
		_posicion, new Rectangle((_tileX * _tamTileX) + (img *
		_tamTileX), 1, _tamTileX, _tamTileY-1), Color.White);
	}
	else if (movimiento == Cliente.ARRIBA)
	{
		j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
		_posicion, new Rectangle((_tileX * _tamTileX) + (img *
		_tamTileX), 51, _tamTileX, _tamTileY-1), Color.White);
	}
	else if (movimiento == Cliente.DERECHA)
```

```
        {
            j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
            _posicion, new Rectangle((_tileX * _tamTileX) + (img *
            _tamTileX), 101, _tamTileX, _tamTileY-1), Color.White);
        }
        else if (movimiento == Cliente.IZQUIERDA)
        {
            j.SpriteBatch.Draw(_textura, _contenedor.Posicion +
            _posicion, new Rectangle((_tileX * _tamTileX) + (img *
            _tamTileX), 151, _tamTileX, _tamTileY-1), Color.White);
            }

        _frameActual++;
}
```

Como se puede observar, al pasarle al SpriteBatch la textura de dibujo, en lugar de pasársela al completo se le indica que se quiere pintar sólo un fragmento (Rectangle) de ella, y es en este rectángulo donde se define el desplazamiento de la coordenada a dibujar, teniéndose que si se quiere dibujar al personaje andando hacia abajo lo encontrará a partir de un desplazamiento de 1 pixel en la coordenada Y, de 51 pixels para ir hacia arriba, de 101 para ir a la derecha y de 151 para ir a la izquierda, como podemos ver en la imagen del tileset del personaje:

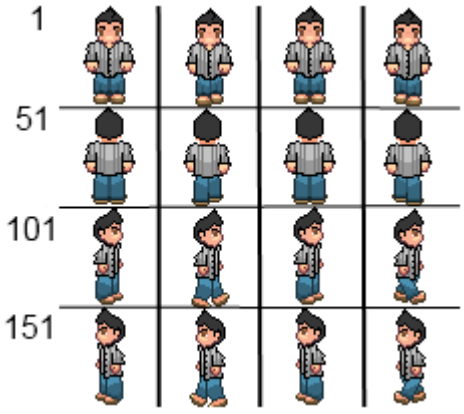

Imagen 45. Coordenadas de desplazamiento en el eje Y del personaje

El otro caso de imagen definida en tileset es el mapa, que carece de animación, por lo que en la clase Mapa sólo se pintará la matriz de tiles[] que contiene los fragmentos de mapa:

```
public void Draw()
{
    for (int i = 0; i < dim_fila; i++)
    {
        for (int j = 0; j < dim_columna; j++)
        {
            tiles[i, j].Draw();
        }
    }
}
```

Este código hace referencia al método Draw() de la clase ImagenTileSet que contiene el siguiente código:

```
public void Draw()
{
      if (!_mostrar)
            return;
      _pantalla.Juego.SpriteBatch.Draw(_textura, _posicion, new
      Rectangle(_tileX * 50, _tileY * 50, 50, 50), Color.White);
}
```

Donde de forma análoga al caso de la representación del personaje, se está indicando que de la textura completa del mapa se quiere representar un fragmento de la imagen de tamaño 50x50 pixels y que está desplazado **_tileX** y **_tileY** respecto al eje de coordenadas.

- **Texto**

Para representar texto en pantalla, en la clase Texto se contará con el siguiente método Draw():

```
public override void Draw()
{
    if (!_mostrar)
        return;
    Juego j = _contenedor.Pantalla.Juego;
    if (_origen.Equals("izquierda"))
    {
        SpriteFont sf = j.Content.Load<SpriteFont>(_estilo);
        j.SpriteBatch.DrawString(sf, _texto, _contenedor.Posicion +
_posicion, _color);
    }
    else if (_origen.Equals("centro"))
    {
        SpriteFont sf = j.Content.Load<SpriteFont>(_estilo);
        Vector2 origen = sf.MeasureString(_texto) * 0.5f;
        j.SpriteBatch.DrawString(sf, _texto, _contenedor.Posicion +
_posicion, _color, 0, origen, 1, SpriteEffects.None, 0);
    }
    else
    {
        throw new Exception("Funcion no valida");
    }
}
```

Este código cargará el SpriteFont para dibujar el texto en función del estilo con el que se instanció el texto, y establecerá las coordenadas de dibujo del texto en función de si se quiere que las coordenadas se calculen desde la esquina superior izquierda del texto o desde el centro de la cadena.

5.2.5.Eventos en la interacción del usuario

Otro aspecto básico a tratar en el desarrollo de un juego es la interacción del usuario con la interfaz gráfica.

En este caso, para el desarrollo de la interfaz se ha optado por utilizar el patrón de diseño Composite, en el que partiendo de objetos simples se puede componer uno más complejo. Para ello todos los elementos simples heredarán de la clase abstracta ComponenteGenerico, la cual a su vez implementará la interfaz Elemento y el patrón de diseño Observer para que al componente contenedor del resto de componentes se propague la interacción del usuario frente a los componentes más sencillos.

Estas clases e interfaces tienen la siguiente codificación:

```
public interface Elemento
{

    Vector2 Posicion
    {
        get;
        set;
    }

    bool Mostrar
    {
        get;
        set;
    }

    string Id
    {
        get;
    }

    /// <summary>
    /// Actualiza la lógica del juego
    /// </summary>
    void Update();

    /// <summary>
    /// Dibuja el juego
    /// </summary>
    void Draw();

    /// <summary>
    /// Devuelve el alto del elemento
    /// </summary>
    /// <returns>Alto total del elemento</returns>
    int ObtenerAlto();

    /// <summary>
    /// Devuelve el ancho del elemento
    /// </summary>
    /// <returns>Ancho total del elemento</returns>
    int ObtenerAncho();
```

```
}

public interface Observer
{
    /// <summary>
    /// Será llamado por los objetos observados para indicar que ha
ocurrido
    /// </summary>
    /// <param name="o">Objeto observado</param>
    /// <param name="evento">Código de evento</param>
    /// <param name="auxiliar">Objeto auxiliar por si hay que aportar
más información</param>
    void Notificar(Observado o, int evento, object auxiliar);

}

public abstract class Observado
{
    public List<Observer> _observadores;
    /// <summary>
    /// Agrega un observador
    /// </summary>
    /// <param name="o">Observador</param>
    public void AgregarObservador(Observer o)
    {
        if (_observadores == null)
        {
            _observadores = new List<Observer>();
        }
        _observadores.Add(o);
    }

    /// <summary>
    /// Elimina un observador
    /// </summary>
    /// <param name="o">Observador</param>
    public void EliminarObservador(Observer o)
    {
        for (int i = 0; i < _observadores.Count; i++)
        {
            if (_observadores[i] == o)
            {
                _observadores.RemoveAt(i);
                i = _observadores.Count;
            }
        }
    }

    /// <summary>
    /// Este método será llamado por los objetos observados para
informar a los observadores
    /// </summary>
    /// <param name="evento">Código de evento</param>
    /// <param name="auxiliar">Objeto para enviar información
adicional</param>
    public void NotificarObservadores(int evento, object auxiliar)
    {
        for (int i = 0; i < _observadores.Count; i++)
        {
            _observadores[i].Notificar(this, evento, auxiliar);
        }
```

```
    }
}
```

Para cerrar el comportamiento del patrón Composite se contará con la clase ContenedorGenerico que implementará la interfaz Elemento con el fin de agrupar los ComponentesGenéricos y a otros ContenedorGenerico, el código de esta clase es el siguiente:

```
public class ContenedorGenerico : Observado, Elemento
{

    protected ContenedorGenerico _contenedor;
    protected Imagen _imagenFondo;
    protected List<Elemento> _elementos;
    protected PantallaGenerica _pantalla;
    protected Vector2 _posicion;
    protected const int ESPACIO = 10;
    protected string _id;

    protected int _ancho;
    protected int _alto;
    protected bool _redimensionado;

    protected bool _cargado;

    protected bool _mostrar;

    protected Elemento ultimo = null;

    /// <summary>
    /// Constructor de contenedores genéricos
    /// </summary>
    /// <param name="pantalla">Pantalla que tiene este
contenedor</param>
    /// <param name="posicion">Posicion de la esquina superior
izquierda del contenedor</param>
    /// <param name="contenedor">Contenedor que contiene a THIS. Null
si es padre.</param>
    public ContenedorGenerico(PantallaGenerica pantalla, Vector2
posicion, ContenedorGenerico contenedor, string id)
    {
        _pantalla = pantalla;
        _posicion = posicion;
        _contenedor = contenedor;
        _elementos = new List<Elemento>();
        _cargado = false;
        _redimensionado = false;
        _mostrar = true;
        _id = id;
    }

    public ContenedorGenerico()
    {
        _elementos = new List<Elemento>();
        _redimensionado = false;
        _cargado = false;
    }

      //Métodos para agregar elementos y ubicarlos en el contenedor
      ...
}
```

Los elementos simples con los que contará la interfaz serán:

- **Imágen**

Codificado en la clase Imagen que hereda de ComponenteGenerico y que ya fue expuesto en el apartado anterior.

- **Texto**

Codificado en la clase Texto que hereda de ComponenteGenerico y que ya fue expuesto en el apartado anterior.

Y los elementos complejos que se generan por composición de los simples y que a su vez también son elementos serán:

- **Botón**

Codificado en la clase Boton que hereda de ComponenteGenerico, estará compuesto por una imagen estática y un texto fijado en su creación.

Los atributos de que se compone esta clase son:

```
// Imagen del botón
private Imagen _imagen;
// Texto del botón
private Texto _texto;
// Indica si el botón ha sido presionado
private bool _presionado;
```

Siendo su constructor:

```
public Boton(ContenedorGenerico c, Vector2 posicion, string id, string
textura, string texto, string estilo, Color color)
    : base(c, posicion, id)
{
    _texto = new Texto(c, new Vector2(posicion.X + 6, posicion.Y + 5),
"texto_" + id, estilo, texto, "izquierda", color);
    _imagen = new Imagen(c, posicion, "imagen_" + id, textura,
_texto.ObtenerAncho() + 15, _texto.ObtenerAlto() + 5);
}
```

Y un ejemplo de instanciación:

```
Boton aceptar = new Boton(res, new Vector2(), "boton_aceptar",
"boton", "Aceptar", "Defecto", Color.White);
```

Por otro lado, para controlar si el botón ha sido presionado se ejecutará el siguiente código en cada ciclo de Update() de la instancia de tipo Boton:

```
public override void Update()
{
    if (!_mostrar)
```

```
            return;
        this.ComprobarSeleccionado();

        if (_seleccionado)
        {
            this.NotificarObservadores(1, this._id);
            _seleccionado = false;
        }
    }

    public override void ComprobarSeleccionado()
    {
        Juego juego = _contenedor.Pantalla.Juego;
        Rectangle areaCubierta = new
    Rectangle((int)(_contenedor.Posicion.X + _posicion.X),
    (int)(_contenedor.Posicion.Y + _posicion.Y),
            this.ObtenerAncho(), this.ObtenerAlto());
        Rectangle clickMouse = new Rectangle(juego.RatonActual.X,
    juego.RatonActual.Y, 1, 1);
        if (juego.RatonActual.LeftButton == ButtonState.Pressed)
        {
            _presionado = true;
        }
        else
        {
            _presionado = false;
        }

        if (clickMouse.Intersects(areaCubierta) &&
    juego.RatonActual.LeftButton != juego.RatonAnterior.LeftButton &&
    _presionado)
        {
            _seleccionado = true;
        }
        else if (!clickMouse.Intersects(areaCubierta) && _presionado)
        {
            _seleccionado = false;
        }
    }
```

Como se puede ver en el código, cuando se interactúa con un botón el valor del atributo heredado _seleccionado se pone a **true**, de modo que en el ciclo de Update() se llama al método NotificarObservadores(…) de la clase Observado para notificar a los observadores asociados a este botón de la acción, y se le pasa al método un identificador con código 1 para que se sepa que la interacción fue con un botón, y para identificar que botón se agrega también la información del identificador del botón.

- **Entrada de texto**

Codificado en la clase EntradaTexto que hereda de ComponenteGenerico y que estará compuesta por una imagen estática cuando el elemento no está seleccionado, otra imagen estática para cuando el elemento esté seleccionado, y un texto que irá cambiando según las pulsaciones de teclas del usuario para la introducción de datos.

Los atributos de esta clase serán:

```
// Imagen de fondo
private Imagen _imagen;
// Texto que se mostrará
private Texto _texto;
// Prompt para pedir el texto
private string _prompt;
// Buffer donde se almacenan los datos antes de mostrarse como texto
private string _buffer;
// Ancho de la entrada de texto
private int _ancho;
// Alto de la entrada de texto
private int _alto;
// Número máximo de caracteres aceptados en la entrada de texto
private int _tamMax = 20;
// Teclas presionadas en el estado anterior, para que no se dupliquen
private List<Keys> _teclasAnteriores;


//Para controlar el aspecto visual al seleccionar
// Textura que se aplicará cuando el componente esté seleccionado
private string _texturaSeleccionada;
// Textura que se aplicará cuando el componente no esté seleccionado
private string _texturaNoSeleccionada;

//Para controlar el parpadeo de |
// Frames que dura el parpadeo
private const int PARPADEO = 30;
// Contabiliza cuantos frames lleva parpadeando
private int _contadorParpadeo;
// Indica si se tiene que mostrar o no el parpadeo
private bool _mostrarParpadeo;
// Indica si el campo será rellenado con **** como una contraseña
private bool _sombreado;
// Guarda lo que está escrito tras el sombreado
private string _bufferSombreada;
```

Su constructor:

```
public EntradaTexto(ContenedorGenerico c, Vector2 posicion, int ancho,
int alto, string id, string texturaSeleccionada,
    string texturaNoSeleccionada, string estilo, string prompt, int
tamMax, bool sombreado, Color color)
    : base(c, posicion, id)
{
    _prompt = prompt;
    _buffer = "";
    _bufferSombreada = "";
    _ancho = ancho;
    _alto = alto;
```

```
    _texturaNoSeleccionada = texturaNoSeleccionada;
    _texturaSeleccionada = texturaSeleccionada;

    _imagen = new Imagen(c, posicion, id, texturaNoSeleccionada,
ancho, alto);
    _texto = new Texto(c, new Vector2(posicion.X, posicion.Y + 1), id,
estilo, _prompt, "izquierda", color);
    _teclasAnteriores = new List<Keys>();
    _mostrarParpadeo = false;
    _contadorParpadeo = 0;
    _tamMax = tamMax;
    _sombreado = sombreado;
}
```

Y un ejemplo de instanciación:

```
EntradaTexto _entrada = new EntradaTexto(c, new Vector2(0,
(_textos.Length + 1) * _textos[0].ObtenerAlto()),
_textos[0].ObtenerAncho(),
_textos[0].ObtenerAlto(),"entrada_texto_chat", "vacio", "vacio",
"defecto", " >", 20, false, Color.Black);
```

El código encargado de que a cada ciclo de Update() la EntradaTexto actualice el texto que ha captura de la interacción del usuario será el siguiente:

```
public override void Update()
{
    if (!_mostrar)
        return;
    this.ComprobarSeleccionado();

    if (_seleccionado)
    {
        _imagen.MiTextura = _texturaSeleccionada;
        //Si está _seleccionado este componente, añadimos la tecla
presionada al buffer de texto
        Juego j = _contenedor.Pantalla.Juego;
        Keys[] teclaPresionada = j.TecladoActual.GetPressedKeys();
        if (j.TecladoActual != j.TecladoAnterior &&
teclaPresionada.Length != 0)
        {
            if (teclaPresionada[0] == Keys.Back)
            {
                //Borramos la última letra introducida
                if (_buffer.Length > 0)
                    _buffer = _buffer.Substring(0, _buffer.Length -
1);
                if (_bufferSombreada.Length > 0)
                    _bufferSombreada = _bufferSombreada.Substring(0,
_buffer.Length - 1);
            }
            else if (teclaPresionada[0] == Keys.Space &&
_buffer.Length < _tamMax)
            {
                if (!_sombreado)
                    _buffer += " ";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += " ";
```

```
                }
            }
            else if (teclaPresionada[0] == Keys.OemComma &&
_buffer.Length < _tamMax)
            {
                if (!_sombreado)
                    _buffer += ",";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += ",";
                }
            }
            else if (teclaPresionada[0] == Keys.OemPeriod &&
_buffer.Length < _tamMax)
            {
                if (!_sombreado)
                    _buffer += ".";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += ".";
                }
            }
            else if (teclaPresionada[0] == Keys.D1 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "1";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "1";
                }
            }
            else if (teclaPresionada[0] == Keys.D2 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "2";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "2";
                }
            }
            else if (teclaPresionada[0] == Keys.D3 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "3";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "3";
                }
            }
            else if (teclaPresionada[0] == Keys.D4 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
```

```
                    _buffer += "4";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "4";
                }
            }
            else if (teclaPresionada[0] == Keys.D5 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "5";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "5";
                }
            }
            else if (teclaPresionada[0] == Keys.D6 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "6";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "6";
                }
            }
            else if (teclaPresionada[0] == Keys.D7 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "7";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "7";
                }
            }
            else if (teclaPresionada[0] == Keys.D8 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "8";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "8";
                }
            }
            else if (teclaPresionada[0] == Keys.D9 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "9";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "9";
                }
```

```
            }
            else if (teclaPresionada[0] == Keys.D0 && _buffer.Length <
_tamMax)
            {
                if (!_sombreado)
                    _buffer += "0";
                else
                {
                    _buffer += "*";
                    _bufferSombreada += "0";
                }
            }
            else if (teclaPresionada[0] == Keys.Enter)
            {
                //Para controlar el ENTER del CHAT
                if (_observadores == null)
                    _observadores = new List<Observer>();

                for (int i = 0; i < _observadores.Count; i++)
                {
                    _observadores[i].Notificar(this, 24,
_texto.MiTexto);
                }
            }
            else if (_buffer.Length < _tamMax &&
!this.caracterProhibido(teclaPresionada[0]))
            {
                //Introducimos la nueva tecla
                List<Keys> lista =
NuevasTeclasPresionadas(teclaPresionada);
                for (int i = 0; i < lista.Count; i++)
                {
                    if (!_sombreado)
                        _buffer += ObtenerTecla(lista[i].ToString());
                    else
                    {
                        _buffer += "*";
                        _bufferSombreada +=
ObtenerTecla(lista[i].ToString());
                    }
                }
            }
        }

        if (_contadorParpadeo == PARPADEO)
        {
            _contadorParpadeo = 0;
            _mostrarParpadeo = !_mostrarParpadeo;
        }

        if (_mostrarParpadeo)
        {
            _buffer += "|";
        }
        AsignarTeclasAnteriores(teclaPresionada);

        _contadorParpadeo++;
    }
    else
    {
        _imagen.MiTextura = _texturaNoSeleccionada;
```

```
    }

    _texto.MiTexto = _prompt + " " + _buffer;

    if (_mostrarParpadeo && _seleccionado)
    {
        //Quitamos el |
        _buffer = _buffer.Substring(0, _buffer.Length - 1);
    }
}

private bool caracterProhibido(Keys keys)
{
    bool res = true;
    string s = keys.ToString();

    if (
        s.Length == 1 ||
        s.Equals("D0") ||
        s.Equals("D1") ||
        s.Equals("D2") ||
        s.Equals("D3") ||
        s.Equals("D4") ||
        s.Equals("D5") ||
        s.Equals("D6") ||
        s.Equals("D7") ||
        s.Equals("D8") ||
        s.Equals("D9")
        )
        res = false;

    return res;
}

public override void ComprobarSeleccionado()
{
    Juego juego = _contenedor.Pantalla.Juego;
    Rectangle areaCubierta = new
Rectangle((int)(_contenedor.Posicion.X + _posicion.X),
(int)(_contenedor.Posicion.Y + _posicion.Y),
        _ancho, _alto);
    Rectangle clickMouse = new Rectangle(juego.RatonActual.X,
juego.RatonActual.Y, 1, 1);
    if (clickMouse.Intersects(areaCubierta) &&
juego.RatonActual.LeftButton == ButtonState.Pressed)
    {
        _seleccionado = true;
        _mostrarParpadeo = true;
    }
    else if (!clickMouse.Intersects(areaCubierta) &&
juego.RatonActual.LeftButton == ButtonState.Pressed)
    {
        _seleccionado = false;
    }
}
```

Como se ve en el código, en primera instancia se comprobará si se ha presionado con el ratón sobre la EntradaTexto, y en caso de que sea así, se parsearán las teclas presionada para mostrar el texto introducido.

- **Chat**

Codificado en la clase Chat que hereda de ComponenteGenerico y que mostrará los últimos 5 textos introducidos por el jugador o por otros jugadores que se encuentren en la partida, y con una EntradaTexto para permitir al jugador local comunicarse con el resto.

Los atributos de esta clase serán:

```
//Textos introducidos y que se mostrarán en pantalla
private Texto []_textos;
//EntradaTexto para que el usuario local envíe mensajes
private EntradaTexto _entrada;
//Pantalla sobre la que está insertado el sistema de chat
private Pantallas.PantallaJuego _pantallaJuego;
```

Siendo su constructor:

```
public Chat(ContenedorGenerico c, Vector2 posicion, String id,
Pantallas.PantallaGenerica pantallaJuego)
    : base(c, posicion, id)
{
    _textos = new Texto[4];

    _pantallaJuego = (Pantallas.PantallaJuego)pantallaJuego;

    Texto aux = new Texto(c, new Vector2(), "aux", "defecto",
"Prueba", "izquierda", Color.White);
    for(int i=0;i<_textos.Length;i++){
        _textos[i] = new Texto(c, new Vector2(7, (i + 1) *
(aux.ObtenerAlto())), "texto_" + i, "defecto", "", "izquierda",
Color.Black);
        _textos[i].MiTexto = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
    }

    _entrada = new EntradaTexto(c, new Vector2(0, (_textos.Length + 1)
* _textos[0].ObtenerAlto()), _textos[0].ObtenerAncho(),
_textos[0].ObtenerAlto(),
        "entrada_texto_chat", "vacio", "vacio", "defecto", " >", 20,
false, Color.Black);

    _entrada.AgregarObservador(this);

    for (int i = 0; i < _textos.Length; i++)
    {
        _textos[i].MiTexto = " ";
    }
}
```

Como se puede observar en el código del constructor del objeto de tipo Chat, a la EntradaTexto _entrada se le agrega como observador la instancia this (ese mismo objeto de Chat), así cuando se interactúe sobre la EntradaTexto esta información le será transmitida a través del patrón Observer al Chat.

Un ejemplo de instanciación será el siguiente:

```
Chat chat = new Chat(res, new Vector2(), "chat", _pantalla);
```

Por otro lado, para enviar los mensajes de chat una vez recibida la notificación de la EntradaTexto con la cadena escrita, se notificará a su vez a los observadores suscritos al objeto chat con el siguiente método:

```
public override void Notificar(Observado o, int evento, object
auxiliar)
{
    for (int i = 0; i < _observadores.Count; i++)
    {
        _observadores[i].Notificar(this, 24, auxiliar);
    }

    //Vaciamos la entrada de texto
    _entrada.MiTexto = "";
}
```

En este caso el observador suscrito para ser notificado de los cambios producidos en chat será la pantalla de juego, que será la encargada de comunicar esta información al cliente TCP para que informe al servidor de juego del mensaje.

Finalmente, para agregar textos al tablón de mensajes de chat, la pantalla haciendo las veces de interfase entre el sistema de comunicaciones cliente/servidor TCP recurrirá al siguiente método:

```
public void ActualizarTextos(string nuevoMsg)
{
    //Comprobamos si el mensaje es para este usuario
    int punto = nuevoMsg.IndexOf('.');
    string destino = nuevoMsg.Substring(0, punto);
    string mensaje = nuevoMsg.Substring(punto+1);
    if (destino.Equals("TODOS") ||
destino.Equals(_pantallaJuego.JugadorLocal.Nombre) ||
mensaje.Contains(_pantallaJuego.JugadorLocal.Nombre+":"))
    {
        _textos[0].MiTexto = _textos[1].MiTexto;
        _textos[1].MiTexto = _textos[2].MiTexto;
        _textos[2].MiTexto = _textos[3].MiTexto;
        _textos[3].MiTexto = mensaje;
    }
}
```

5.2.6.Sistema de colisiones

Otro aspecto de obligado tratado en el desarrollo de un videojuego es el sistema de colisiones a emplear.

Este sistema será el encargado de controlar que el jugador no ocupe una posición que no esté habilitada para ello como pueden ser muros, objetos en el suelo u otros jugadores.

Para ello, durante el ciclo de ejecución del juego, en el método Update(), una vez se haya detectado la interacción del usuario con el juego como se explicó en el apartado 5.2.2. Movimiento del jugador y se sepa en qué dirección va a realizar el movimiento, antes de que se actualicen las coordenadas del jugador será necesario comprobar si es posible que ocupe esta posición.

En este caso de ejemplo se establecerá que el jugador no puede ocupar las posiciones definidas por objetos del mapa tales como árboles, casas, fuentes y posiciones en las que se encuentren otros jugadores.

La forma en que se controlará si el jugador colisiona con estos objetos tendrá que tener en cuenta tres factores:

- La dirección del movimiento establecerá en qué coordenadas habrá que sumar y restar posiciones, ya que si por ejemplo se desplaza hacia abajo las coordenadas del eje Y crecerán, mientras que si se desplaza hacia arriba las coordenadas del eje Y decrecerán.
- La coordenada del jugador se establece como la posición central del gráfico del jugador, de modo que cuando se calcule si se colisiona con un objeto, a la coordenada actual del jugador habrá que sumarle o restarle (según la dirección del movimiento) la mitad del tamaño del gráfico del jugador.
- Teniendo en cuenta los dos puntos anteriores se tendría la coordenada de un extremo del gráfico que en el momento del cálculo puede no haber colisionado, pero al actualizar la coordenada en función de la velocidad de desplazamiento en caso de que no se produjera colisión en el cálculo podría introducir al gráfico del personaje en unas coordenadas ocupadas por un objeto que produce colisión y del que por tanto no podrá salir.

De estos tres factores se deduce que:

- Será necesario sumar o restar a la coordenada X la mitad del ancho del gráfico y a la coordenada Y la mitad del alto del gráfico en función del movimiento.
- A el cálculo anterior habrá que sumar o restar en función del movimiento, el valor del desplazamiento que realizaría en caso de que no colisionara, para comprobar si esta posición es una posición no válida y no permitirle realizar el movimiento volviendo a establecer las coordenadas de antes de realizar el cálculo.

Finalmente y como se indicaba, en el método Update() del ciclo de ejecución habrá que introducir el siguiente código:

```
public override void Update()
{
      //Código para la detección del movimiento del jugador

      if (_mapa.Colision(mov))
      {
          colision = true;
          _jugadorLocal.FinalizarMovimiento(1,
      _jugadorLocal.PosicionMovimiento);
      }
      else if (ColisionJugadores(mov))
      {
          colision = true;
          _jugadorLocal.FinalizarMovimiento(1,
      _jugadorLocal.PosicionMovimiento);
      }
      else
      {
          Vector2 posicion = _jugadorLocal.PosicionMovimiento;
          _jugadorLocal.FinalizarMovimiento(mov, posicion);
      }

      //Código para el cálculo de posiciones de jugadores y mapas
      . . .

}
```

Donde el método de detección de colisión con el mapa y que estaría ubicado en la clase Mapa sería:

```
public bool Colision(int mov)
{
    Jugador j = pantalla.JugadorLocal;
    int x = 0;
    int y = 0;
    Vector2 coordenadas = j.PosicionMovimiento;

    int desvix = (j.Grafico.ObtenerAncho() - 50) / 2;
    int desviy = (j.Grafico.ObtenerAlto() - 50) / 2;

    Vector2 pos = j.Grafico.Posicion;
    int width = j.Grafico.ObtenerAncho();
    int height = j.Grafico.ObtenerAlto();

    switch (mov)
    {
        case Cliente.ARRIBA:
            {
                x = (int)coordenadas.X - desvix;
                y = (int)coordenadas.Y + 10 + desviy;
                break;
            }
        case Cliente.ABAJO:
            {
                x = (int)coordenadas.X - desvix;
                y = (int)coordenadas.Y + 30 + desviy;
```

```
                break;
            }
        case Cliente.IZQUIERDA:
            {
                x = (int)coordenadas.X - 5 - desvix;
                y = (int)coordenadas.Y + 20 + desviy;
                break;
            }
        case Cliente.DERECHA:
            {
                x = (int)coordenadas.X + 20 - desvix;
                y = (int)coordenadas.Y + 20 + desviy;
                break;
            }
    }

    int celdax = x / 50;
    int celday = y / 50;

    try
    {
        if (x <= 0 || y <= 0)
            return true;
        else if (logica[celday, celdax] == 1)
            return true;
        else
            return false;
    }
    catch
    {
        return true;
    }
}
```

Como se puede ver en este fragmento de código, al tener la matriz lógica[] en la clase Mapa la cual almacena un 1 para las posiciones del mapa que van a producir colisión, sólo es necesario calcular la futura posición del jugador y comprobar si en esta matriz esa posición es un 1, devolviendo **true** e indicando así la colisión.

Además se introduce la comprobación dentro de un bloque **try…catch** para controlar el caso en que se esté calculando que se sale del mapa, el cual daría un error en tiempo de ejecución al acceder a una posición de la matriz inexistente.

Por último, el método para la detección de colisiones con otros jugadores que estaría ubicado en la misma clase que el método Update() sería:

```
private bool ColisionJugadores(int mov)
{
    Jugador j = _jugadorLocal;
    int x = 0;
    int y = 0;

    switch (mov)
    {
        case Cliente.ARRIBA:
            {
                x = (int)j.PosicionMovimiento.X;
                y = (int)j.PosicionMovimiento.Y - 10;
                break;
```

```
            }
        case Cliente.ABAJO:
            {
                x = (int)j.PosicionMovimiento.X;
                y = (int)j.PosicionMovimiento.Y + 40;
                break;
            }
        case Cliente.IZQUIERDA:
            {
                x = (int)j.PosicionMovimiento.X - 20;
                y = (int)j.PosicionMovimiento.Y;
                break;
            }
        case Cliente.DERECHA:
            {
                x = (int)j.PosicionMovimiento.X + 30;
                y = (int)j.PosicionMovimiento.Y;
                break;
            }
    }

    Rectangle local = new Rectangle(x - 25, y - 25, 50, 50);
    for (int i = 0; i < _jugadores.Count; i++)
    {
        if (!_jugadores[i].Local)
        {
            Rectangle otro = new
            Rectangle((int)_jugadores[i].PosicionMovimiento.X - 25,
            (int)_jugadores[i].PosicionMovimiento.Y - 25, 50, 50);
            if (local.Intersects(otro))
                return true;
        }
    }

    return false;
}
```

Como se puede observar el código de detección difiere de la detección con objetos del mapa, esto es debido a que los personajes pueden estar en movimiento y habría que tener en cuenta no sólo la dirección de movimiento del jugador local, sino también la del resto de jugadores.

Para ello se usará el método Intersects(Rectangle otro) de la clase Rectangle, el cual permite comprobar si un rectángulo intersecta con otro, por lo que se crearán dos rectángulos con las coordenadas de posible colisión y si estas producen colisión se devolverá **true**.

5.2.7.Envío de la información del jugador

Al tratarse de un juego MMO será necesario comunicar la información como ya se estudió en los diferentes apartado del punto 4 Manual de desarrollo de infraestructura MMO.

Para facilitar la transmisión de datos a través de la red, se ha desarrollado la clase Paquete que encapsulará los datos en su envío y ayudará a su reconstrucción a la llegada en el otro extremo.

La clase Paquete consta de los siguientes atributos:

```
//Array de bytes con los datos a enviar
private byte[] datos;
//Guarda los elementos, según el tipo de datos que sean (int o string)
private List<Object> elementos;
//Guarda los tipos de datos de cada elemento (0 para enteros, 1 para
strings)
private List<int> tipos;
```

La idea que persigue esta clase es enviar un flujo de bytes con el siguiente formato:

[num_bytes_mapa][mapa] - [[num_bytes_elemento][elemento]]*

Donde cada campo significa:

- **num_bytes_mapa**

Contiene un entero codificado en un byte que indicará el tamaño del siguiente campo, que también será el número de elementos (enteros o cadenas) que contiene el paquete enviado.

Por ejemplo, si num_bytes_mapa = 4, tendremos un campo mapa de 4 bytes y el paquete contendrá 4 elementos

- **mapa**

Contiene una sucesión de enteros codificados en un byte que indican en caso de valer 0 que esa posición del mapa es un entero, y en caso de valer 1 que esa posición del mapa es una string.

Por ejemplo, si mapa = 0010, tendremos un primer elemento entero, un segundo elemento entero, un tercer elemento string y un cuarto elemento entero.

- **num_bytes_elemento**

Indica que el elemento que se va a leer ocupa X bytes.

Por ejemplo, si num_bytes_elemento = 5, el elemento que se va a leer a continuación estará compuesto por 5 bytes.

- **elemento**

Este es el elemento que se está leyendo, que será entero o string en función de si esa posición del mapa era un 0 o un 1, y que ocupará tantos caracteres como índice num_bytes_elemento. Dado que lo que se recibe es un array de bytes, será necesario realizar una conversión una vez escogidos los bytes del elemento que variará en función del tipo de dato transmitido.

Para las strings se hará la conversión del valor numérico ASCII de cada byte para saber que letra se transmitió.

Por ejemplo: byte[0]=72,byte[1]=111,byte[2]=108,byte[3]=97 será traducido como "Hola"

Para los enteros, primero se comprobará si hay un 1 o un 0 que indicará si el valor es negativo o positivo respectivamente, y después se cogerán los bytes agrupando cada byte dos números.

Por ejemplo: byte[0]=0, byte[1]=52, byte[2]=0, byte[3]=1 será traducido como 520001

Para realizar esta funcionalidad la clase Paquete constará de dos constructores, uno para Paquetes que se van a enviar y de los que hay que construir el array de bytes a través de strings y enteros, y otro para Paquetes que se reciben como un array de bytes y de los que hay que construir los strings y enteros. Los constructores son los siguientes:

```
public Paquete()
{
    datos = new byte[0];
    elementos = new List<Object>();
    tipos = new List<int>();
}

public Paquete(byte[] datos)
{
    this.datos = datos;
    elementos = new List<Object>();
    tipos = new List<int>();

    this.Desempaquetar();
}
```

Donde Desempaquetar() realizará:

```
public void Desempaquetar()
{
    int indice = 0;
    int dim;
    //[num_bytes_mapa]
    int iteraciones = (int)datos[indice];
    indice++;
    //[mapa]
    for (int i = 0; i < iteraciones; i++)
    {
        tipos.Add((int)datos[indice + i]);
    }
    indice += iteraciones;
    for (int i = 0; i < iteraciones; i++)
    {
        if (tipos[i] == 0) //Si el elemento es un entero
        {
            int n;
            //[num_bytes_elemento]
```

```
            dim = (int)datos[indice];
            indice++;
            int negativo = (int)datos[indice];
            indice++;
            //[elemento]
            n = ObtenerInt(indice, dim);
            indice += dim;
            if (negativo == 0) //No es negativo
            {
                elementos.Add(n);
            }
            else if (negativo == 1) //Es negativo
            {
                elementos.Add(n * -1);
            }
        }
        else if (tipos[i] == 1) //Si el elemento es un string
        {
            string s;
            //[num_bytes_elemento]
            dim = (int)datos[indice];
            indice++;
            //[elemento]
            s = ObtenerString(indice, dim);
            indice += dim;
            elementos.Add(s);
        }
    }
}
```

Para el caso de agregar datos al Paquete que se quiere enviar, se tendrán los siguientes dos métodos:

```
public void AgregarElemento(int n)
{
    //Insertamos el tipo de elemento en la lista de tipos (int = 0)
    tipos.Add(0);
    //Insertamos el elemento en la lista de elementos
    elementos.Add(n);
}

public void AgregarElemento(string s)
{
    //Insertamos el tipo de elemento en la lista de tipos (string = 1)
    tipos.Add(1);
    //Insertamos el elemento en la lista de elementos
    elementos.Add(s);
}
```

Y cuando se quiera enviar el paquete se accederá a la propiedad Datos del Paquete, que realizará:

```
public byte[] Datos
{
    get
    {
        datos = new byte[0];
        this.Empaquetar();
        return datos;
```

```
    }
}

public void Empaquetar()
{
    int dim;
    //[num_bytes_mapa]
    this.AnadirEnteroSinSigno(tipos.Count);
    //[mapa]
    for (int i = 0; i < tipos.Count; i++)
    {
        this.AnadirEnteroSinSigno(tipos[i]);
    }
    //Guardamos los elementos
    for (int i = 0; i < elementos.Count; i++)
    {
        if (tipos[i] == 0) //Si el elemento es un entero
        {
            int valor = (int)elementos[i];
            //[num_bytes_elemento]
            dim = this.ObtenerDimension(valor);
            this.AnadirEnteroSinSigno(dim);
            //[elemento]
            this.AnadirEnteroConSigno(valor);
        }
        else if (tipos[i] == 1) //Si el elemento es una string
        {
            string valor = (string)elementos[i];
            this.AnadirEnteroSinSigno(valor.Length);
            this.AnadirString(valor);
        }
    }
}

private void AnadirEnteroSinSigno(int n)
{
    string aux = n.ToString();
    int indice = 0;
    if (aux.Length % 2 == 1)
    {
        //si es impar metemos un 0 a la izquierda
        aux = "0" + aux;
    }
    int dim = (int)(aux.Length * 0.5);
    byte[] res = new byte[dim];

    for (int i = 0; i < aux.Length; i += 2)
    {
        res[indice] = (byte)(Int32.Parse(aux.Substring(i, 2)));
        indice++;
    }

    this.Anadir(res);
}

private void AnadirEnteroConSigno(int n)
{
    string aux = n.ToString();
    bool negativo = false;
    if (aux.Contains("-"))
    {
```

```
            negativo = true;
            aux = aux.Substring(aux.IndexOf('-') + 1);
        }
        int indice = 0;
        if (aux.Length % 2 == 1)
        {
            //si es impar metemos un 0 a la izquierda
            aux = "0" + aux;
        }
        int dim = (int)(aux.Length * 0.5);
        byte[] res = new byte[dim];

        for (int i = 0; i < aux.Length; i += 2)
        {
            res[indice] = (byte)(Int32.Parse(aux.Substring(i, 2)));
            indice++;
        }

        if (negativo)
        {
            this.Anadir((byte)1);
        }
        else
        {
            this.Anadir((byte)0);
        }
        this.Anadir(res);
    }

    private void AnadirString(string cadena)
    {
        byte[] res = Encoding.ASCII.GetBytes(cadena);
        this.Anadir(res);
    }

    private void Anadir(byte[] datos)
    {
        byte[] aux = new byte[this.datos.Length + datos.Length];
        for (int i = 0; i < this.datos.Length; i++)
        {
            aux[i] = this.datos[i];
        }
        for (int i = 0; i < datos.Length; i++)
        {
            aux[this.datos.Length + i] = datos[i];
        }
        this.datos = aux;
    }
```

Establecida la información que se va a transmitir, queda pendiente tratar cómo se va a transmitir, por ello y debido a que al tratarse de un juego es indispensable que mientras se envía y recibe información no se puede bloquear la ejecución, se recurrirá al uso de concurrencia, por lo que se tendrán tres hilos de ejecución:

- Hilo de juego: Se encargará de seguir el ciclo de ejecución del juego para actualizar coordenadas de jugadores, detectar movimientos, representar los gráficos en pantalla, etc.
- Hilo de envío: Recibirá los paquetes que se quieren enviar desde el hilo del juego al servidor para que éste los comunique al resto de jugadores no locales.
- Hilo de recepción: Recibirá los paquetes del servidor y se los comunicará al hilo de juego para que actualice la lógica de los jugadores que no son locales.

Los hilos de envío y recepción harán uso de la clase InterfazCliente, la cual hará las veces de interfaz entre las clases encargadas de dirigir el juego (clase principal Juego y diferentes clases Pantalla) y la clase Cliente encargada de enviar y recibir información del servidor.

El objeto InterfazCliente será instanciado durante la creación del objeto de la clase principal Juego la cual tendrá los siguientes atributos para controlar la comunicación:

```
//Interfaz de comunicacion
private InterfazCliente _cliente;
/**** Posibles valores para _estadoComunicacion:
* -2. Error.
* -1. En espera de respuesta.
* 0. Sin conexión establecida.
* 1. Conectado con SL, esperando a enviar usuario y contraseña a SL.
* 2. Conectado con SL, esperando a selección, borrado o creación de
personaje.
* 3. Conectado con SPJ.
****/
private int _estadoComunicacion;
```

Teniendo las siguientes líneas de código en su constructor:

```
public Juego()
{
      graphics = new GraphicsDeviceManager(this);
      Content.RootDirectory = "Content";

      miTecladoActual = new KeyboardState();
      miTecladoAnterior = new KeyboardState();
      miRatonActual = new MouseState();
      miRatonAnterior = new MouseState();
      _controles = new Controles();

      miNivel = 0;
      miPantallaCargada = false;

      _cliente = new InterfazCliente(this);
      _estadoComunicacion = 0;
```

```
        _listaPersonajes = new List<string>();
}
```

De modo que para arrancar los hilos de recepción y envío de información, será necesario de forma previa instancia un objeto de tipo InterfazCliente, el cual tiene los siguientes atributos:

```
//Instancia de cliente, encargado de enviar y recibir información del
SPJ
private Cliente _c;
//Nombre de _usuario (login)
private string _usuario;
//Contraseña (login)
private string _passwd;
//Identificador del personaje seleccionado
private int _id_pj;
//Identificador de cliente en SL
private int _id_sl;
//Identificado de cliente en SPJ
private int _id_spj;
//Instancia de tipo Jugador con el personaje local
private Jugador _local;
//Nombre del personaje local
private string _nombrePersonajeLocal;
//Lista de jugadores conectados a la partida
private List<Jugador> _jugadores;
//Instancia de la clase principal Juego
private Juego _juego;
```

Siendo su constructor:

```
public InterfazCliente(Juego juego)
{
        _c = new Cliente(this);
        _jugadores = new List<Jugador>();
        _juego = juego;
}
```

Una vez se tenga la instancia de la InterfazCliente y dado que el hilo de juego es el hilo principal de ejecución, será desde la clase encargada de dirigir la partida desde donde se lancen los otros dos hilos de ejecución.

Con todos estos elementos instanciados y preparados para comunicar información, es cuando empieza el protocolo establecido en el apartado 4.2.4. Protocolo de transporte, flujo de datos entre servidor y clientes.

De forma esquemática este flujo consistirá en:

1. Conectar con el servidor de conexión y obtener las direcciones IP y puerto de servidor de los servidores de login y de juego
2. Conectar con el servidor de login, realizar un proceso de login y seleccionar el personaje con el que jugar
3. Conectar con el servidor de juego y comenzar la partida

- **Negociación con el servidor de conexión**

Para la resolución del punto 1, y como la ejecución del juego se dividirá en pantallas según las posibilidades de interacción del usuario, se introducirá el siguiente código en el constructor de la clase PantallaLogin que después de la pantalla con el logo del juego (clase PantallaAnimación) será la primera con la interactuará el jugador:

```
public PantallaLogin(Juego juego): base(juego)
{
      Interfaz.InterfazCliente c = _juego.Cliente;
      _hiloAuxiliar = new Thread(new ThreadStart(c.ConectarSC));
      _hiloAuxiliar.Start();
      _juego.AgregarHilo(_hiloAuxiliar);
      _juego.EstadoComunicacion = -1;
}
```

Al heredar la PantallaLogin de la clase abstracta PantallaGenérica, de forma automática se habrá añadido la instancia de PantallaLogin como un observador de la instancia InterfazCliente, y esto ocurre por el código y atributos del constructor de PantallaGenérica que son los siguientes:

```
      // Contenedores que formarán la interfaz
      protected List<ContenedorGenerico> _interfaz;
      // Instancia de juego
      protected Juego _juego;

public PantallaGenerica(Juego juego)
{
    _juego = juego;
    _interfaz = new List<ContenedorGenerico>();

    //Agregamos la pantalla como observador de la interfaz de
comunicacion
    _juego.Cliente.AgregarObservador(this);
    //Le decimos a la pantalla que le envíe las notificaciones al
juego
    this.AgregarObservador(_juego);
}
```

Volviendo al código del constructor de PantallaLogin, hace referencia al método ConectarSC() de la clase InterfazCliente desde un nuevo hilo de ejecución, esto se hace de este modo para evitar que la comunicación con el servidor de conexión bloquee el hilo de ejecución del juego, y el método ConectarSC()hará lo siguiente:

```
public void ConectarSC()
{
    string s = _c.ConectarSC();
    NotificarObservadores(20, s);
}
```

Este código llamará al método ConectarSC() de la clase Cliente, que realizará:

```
public string ConectarSC()
{
    byte[] data;
```

```
    IPHostEntry address = Dns.GetHostEntry(host);
    IPEndPoint ipep = new IPEndPoint(Soporte.DevolverIPv4(address),
puertoTcpEnvio);
    Socket server = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    server.NoDelay = true;

    try
    {
        server.Connect(ipep);
    }
    catch (SocketException e)
    {
        return "Servidor no encontrado";
    }

    //Conexión (comando 1)
    Paquete pg = new Paquete();
    pg.AgregarElemento(1);
    server.Send(pg.Datos);
    data = new byte[BYTES];
    int recv = server.Receive(data);
    Paquete ps = new Paquete(data);
    #region Distribución de datos para "data"
    //[0] = se pudo (1) o no (0) realizar la conexion
    //[1] = IP de SL
    //[2] = Puerto por el que nos atiende SL
    //[3] = IP de SPJ
    //[4] = Puerto por el que nos atiende SPJ
    #endregion

    if ((int)ps.ObtenerElemento(0) == 0)
    {
        return "No se pudo realizar la conexión porque no están los
servidores preparados";
    }
    else
    {
        #region Recogemos los datos de SL
        //Cogemos la dirección IP del SL
        IP_SL =
Dns.GetHostAddresses((string)ps.ObtenerElemento(1))[0];
        //Cogemos el puerto del SL
        PUERTO_SL = (int)ps.ObtenerElemento(2);
        #endregion
        #region Recogemos los datos de SPJ
        //Cogemos la dirección IP del SPJ
        IP_SPJ =
Dns.GetHostAddresses((string)ps.ObtenerElemento(3))[0];
        //Cogemos el puerto del SPJ
        PUERTO_SPJ = (int)ps.ObtenerElemento(4);
        #endregion
    }

    server.Shutdown(SocketShutdown.Both);
    server.Close();

    string res = "Se ha conectado con el servidor SC. SL=" + IP_SL +
":" + PUERTO_SL + ". SPJ=" + IP_SPJ + ":" + PUERTO_SPJ;
    return res;
}
```

En este código se puede observar de qué forma se realiza la conexión con el servidor de conexión, controlando si falla y devolviendo esa información en caso de que suceda, enviándole un comando 1 de petición de conexión de cliente y devolviendo una cadena de texto con la información de las IP y puerto de los servidores de login y juego.

Volviendo al código de ConectarSC() de la InterfazCliente, la segunda línea de código informa a todos los observadores de la string que se ha generado en ConectarSC() de Cliente.

En este caso como el observador es la clase PantallaLogin, llegará a su método Notificar (…) el evento número 20 con la cadena de texto generada. De modo que este método Notificar (…) tendrá el siguiente código:

```
public override void Notificar(Observado o, int evento, object
auxiliar)
{
	if (evento == 20)
	{
		List<Elemento> lista = this._interfaz[2].Elementos;
		Texto sc = (Texto)lista[0];
		if (((string)auxiliar).Contains("conectado"))
		{
			sc.MiTexto = "Estado del SC: Activo y funcionando";
			_hiloAuxiliar = new Thread(new
			ThreadStart(_juego.Cliente.ConectarSL));
			_hiloAuxiliar.Start();
			_juego.AgregarHilo(_hiloAuxiliar);
			return;
		}
		sc.MiTexto = "Estado del SC: " + (string)auxiliar;
	}
}
```

En este código se puede ver como si en la cadena notificada por el elemento observado (InterfazCliente) contiene la palabra “conectado”.

En caso de que no la tuviera, la PantallaLogin sólo indicaría el mensaje que podría ser o bien que no ha encontrado al servidor de conexión, o que no estaban preparados los servidores necesarios para comenzar la partida.

El otro caso es que contenga la palabra “conectado”, que dará paso al punto de negociación de conexión con el servidor de login.

- **Negociación de conexión con el servidor login**

El otro caso es que contenga la palabra “conectado”, caso en el que creará un nuevo hilo de ejecución sobre el que realizará la petición de conexión en el servidor de login y de donde obtendrá su identificador en este servidor. El código de esta funcionalidad comienza en la llamada desde la InterfazCliente al método ConectarSL() que realizará lo siguiente:

```
public void ConectarSL()
{
    _id_sl = _c.ConectarSL();
    NotificarObservadores(21, _id_sl);
}
```

De forma análoga al caso de la conexión con el servidor de conexión, se realizará una llamada al método ConectarSL() de la clase Cliente, que realizará:

```
public int ConectarSL()
{
    byte[] data = new byte[BYTES];
    IPHostEntry address = Dns.GetHostEntry(IP_SL);
    IPEndPoint ipep = new IPEndPoint(Soporte.DevolverIPv4(address),
PUERTO_SL);
    envioSL = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    try
    {
        envioSL.Connect(ipep);
    }
    catch (SocketException e)
    {
        return -1;
    }

    //Enviamos el SPJ que nos antenderá
    Paquete pg = new Paquete();
    pg.AgregarElemento(IP_SPJ.ToString());
    envioSL.Send(pg.Datos);

    //Recibimos nuestro identificador del SL
    envioSL.Receive(data);
    pg = new Paquete(data);
    int id_sl = (int)pg.ObtenerElemento(0);

    return id_sl;
}
```

Este código intentará conectar con el servidor de login y realizará la comunicación como se definió en el apartado 4.2.4. Protocolo de transporte, flujo de datos entre servidor y clientes.

Finalmente, con el identificador obtenido por parte del servidor de login, se volverá a la segunda línea de código del método ConectarSL() de la clase InterfazCliente la cual notificará a los observadores (PantallaLogin) el identificador a través del evento número 21 que ampliará el código de Notificar():

```
public override void Notificar(Observado o, int evento, object
auxiliar)
{
	if (evento == 20)
	{
		List<Elemento> lista = this._interfaz[2].Elementos;
		Texto sc = (Texto)lista[0];
		if (((string)auxiliar).Contains("conectado"))
		{
			sc.MiTexto = "Estado del SC: Activo y funcionando";
			_hiloAuxiliar = new Thread(new
			ThreadStart(_juego.Cliente.ConectarSL));
			_hiloAuxiliar.Start();
			_juego.AgregarHilo(_hiloAuxiliar);
			return;
		}
		sc.MiTexto = "Estado del SC: " + (string)auxiliar;
	}
	else if (evento == 21)
	{
		int id_sl = (int)auxiliar;
		if (id_sl != -1)
			_juego.EstadoComunicacion = 1;
		else
			_juego.EstadoComunicacion = -2;
	}
}
```

Este código actualizará el estado de comunicación del juego y permitirá que el usuario realice un login.

Para ello introducirá los datos de usuario y contraseña en la interfaz:

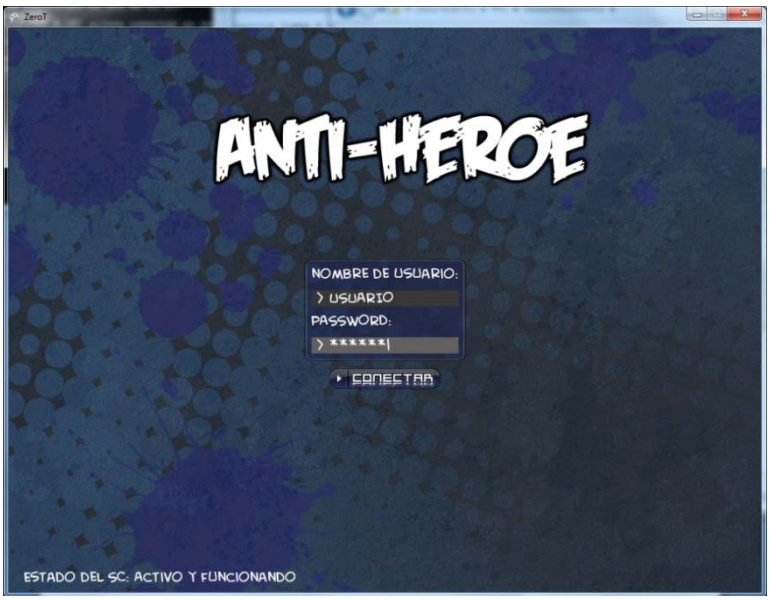

Imagen 46. Pantalla de Login de AntiHeroe

Y a continuación presionará el botón de conectar.

Estos elementos donde introduce el usuario sus datos son diferentes objetos de tipo ComponenteGenérico agregados a un contenedor con el siguiente código:

```
public ContenedorGenerico CrearContenedorLogin()
{
    ContenedorGenerico res = new ContenedorGenerico(_pantalla, new
Vector2(), null, "contenedor_login");

    Texto tUsuario = new Texto(res, new Vector2(), "texto_usuario",
"Defecto", "nombre de usuario:", "izquierda", Color.White);

    EntradaTexto etUsuario = new EntradaTexto(res, new Vector2(),
tUsuario.ObtenerAncho(), tUsuario.ObtenerAlto(),
        "entradatexto_usuario", "FondoGrisSeleccionado",
"FondoGrisNoSeleccionado", "Defecto", " >", 13, false);

    Texto tPasswd = new Texto(res, new Vector2(), "texto_passwd",
"Defecto", "PASSWORD:", "izquierda", Color.White);

    EntradaTexto etPasswd = new EntradaTexto(res, new Vector2(),
tUsuario.ObtenerAncho(), tUsuario.ObtenerAlto(),
        "entradatexto_passwd", "FondoGrisSeleccionado",
"FondoGrisNoSeleccionado", "Defecto", " >", 13, true);

    Boton boton1 = new Boton(res, new Vector2(), "boton_conectar",
"BotonConectar");
    boton1.AgregarObservador(_pantalla);

    res.AgregarElemento(tUsuario);
    res.AgregarElemento(etUsuario, "abajo", "izquierda", true);
    res.AgregarElemento(tPasswd, "abajo", "izquierda", true);
    res.AgregarElemento(etPasswd, "abajo", "izquierda", true);
    res.AgregarFondo("FondoAzulTransparente", true);
    res.AgregarElemento(boton1, "abajo", "centro", true);

    res.SituarCentro(new Vector2(_pantalla.Juego.Width * 0.5f,
_pantalla.Juego.Height * 0.5f));

    return res;
}
```

Donde como se puede ver en el código, al botón de Conectar le ha sido agregado un observador que es la PantallaLogin para que cuando el jugador interactúe con el ratón sobre él esta información le llegue a la PantallaLogin.

Se generará por tanto un evento que llegará al método Notificar() de PantallaLogin y que ampliará el contenido de este método de la siguiente forma:

```
public override void Notificar(Observado o, int evento, object
auxiliar)
{
      if (evento == 1)
      {
            Boton b = (Boton)o;
            switch (b.Id)
            {
                  case "boton_conectar":
```

```
                {
                        if (_juego.EstadoComunicacion == 1)
                        {
                                Interfaz.InterfazCliente c =
                                _juego.Cliente;
                                //Cogemos los datos introducidos por el
                                usuario en el login
                                List<Elemento> lista =
                                this._interfaz[1].Elementos;
                                string nombre =
                                ((EntradaTexto)lista[1]).MiTexto;
                                string pass =
                                ((EntradaTexto)lista[3]).MiTexto;
                                List<string> nombresPjs =
                                c.EnviarUsuarioSL(nombre, pass);
                                if (nombresPjs.Count > 0)
                                {
                                        if (nombresPjs[0].Equals("LOGIN
                                        INCORRECTO"))
                                        {
                                                ConstructorContenedor co =
                                                new
                                                ConstructorContenedor(this);
                                                _mensajes =
                                                co.CrearContenedorMensajeLogi
                                                nIncorrecto();
                                                _mensajes.Posicion = new
                                                Vector2((_juego.Width * 0.5f)
                                                - (_mensajes.ObtenerAncho() *
                                                0.5f),
                                                (_juego.Height * 0.5f) -
                                                (_mensajes.ObtenerAlto() *
                                                0.5f));
                                                return;
                                        }
                                }
                                _juego.EstadoComunicacion = 1;

                                //Tenemos la lista de personajes, se la
                                devolvemos al juego para que cree la
                                siguiente pantalla
                                NotificarObservadores(22, nombresPjs);
                        }
                        break;
                }
        }
}
else if (evento == 20)
{
        List<Elemento> lista = this._interfaz[2].Elementos;
        Texto sc = (Texto)lista[0];
        if (((string)auxiliar).Contains("conectado"))
        {
                sc.MiTexto = "Estado del SC: Activo y funcionando";
                _hiloAuxiliar = new Thread(new
                ThreadStart(_juego.Cliente.ConectarSL));
                _hiloAuxiliar.Start();
                _juego.AgregarHilo(_hiloAuxiliar);
                return;
        }
        sc.MiTexto = "Estado del SC: " + (string)auxiliar;
```

```
        }
        else if (evento == 21)
        {
                int id_sl = (int)auxiliar;
                if (id_sl != -1)
                        _juego.EstadoComunicacion = 1;
                else
                        _juego.EstadoComunicacion = -2;
        }
}
```

En este código, al presionar sobre el botón Conectar se comprueba si el estado de las comunicaciones es igual a 1, es decir, si se está conectado con el SL, y en ese caso se envían los datos del nombre de usuario y contraseña introducidos haciendo uso del método EnviarUsuarioSL(nombre, pass) de la InterfazCliente, el cual realizará lo siguiente:

```
public List<string> EnviarUsuarioSL(string _usuario, string _passwd)
{
    this._usuario = _usuario;
    this._passwd = _passwd;

    List<string> res = new List<string>();
    Paquete pg = new Paquete();
    pg.AgregarElemento(_usuario);
    pg.AgregarElemento(_passwd);
    Paquete aux = _c.EnviarSL(pg);

    int resultado = (int)aux.ObtenerElemento(0);
    if (resultado == 0)
    {//Login incorrecto
        res.Add("LOGIN INCORRECTO");
    }
    else if (resultado == 1)
    {
        int count = aux.ObtenerCountElementos();
        for (int i = 1; i < count; i++)
        {
            res.Add((string)aux.ObtenerElemento(i));
        }
    }
    return res;
}
```

Como se puede ver, envía el nombre de usuario y contraseña y como se definió en el flujo de datos, en caso de que reciba un 0 es que el usuario y contraseña no son correctos por lo que se informará al jugador, y en caso de que se reciba un 1, adicionalmente se habrá agregado una lista con los jugadores disponibles para el usuario, jugadores que serán agregados a una lista y devueltos al método Notificar() de PantallaLogin.

Una vez llega la lista de jugadores a PantallaLogin, ésta se los notifica a sus elementos observadores que en este caso es la clase principal Juego la cual tendrá el siguiente código:

```
public void Notificar(Observado o, int evento, object auxiliar)
{
    if (evento == 22)
```

```
    //Se lanza cuando se obtienen los nombres de los personajes del
servidor SL en PantallaLogin
    {
        _listaPersonajes = (List<string>)auxiliar;
        //Pasamos al nivel de selección de personaje
        PantallaSiguiente();
    }
}
```

Con la lista de personajes en el control de la clase Juego se pasa a la siguiente pantalla que será la clase PantallaSeleccionPJ donde el jugador seleccionará su personaje enviando este comando al servidor de login. Los atributos y constructor de esta clase son:

```
List<string> _listaPersonajes;

public PantallaSeleccionPj(Juego juego, List<string> listaPersonajes)
: base (juego)
{
      _listaPersonajes = listaPersonajes;
}
```

Una captura del conjunto de elementos que presenta esta pantalla es la siguiente:

Imagen 47. Pantalla de selección de personaje de AntiHeroe

Como se puede observar en la imagen, la pantalla cuenta con unos contenedores con la información y acciones que el usuario puede realizar respecto a sus personajes, estos elementos están agregados a su interfaz de elementos y son de tipo ContenedorSeleccionPersonaje que heredarán de ContenedorGenerico el cual implementará la interfaz Elemento para poder ser agregado como elementos de la interfaz gráfica.

En cuanto al ContenedorGenerico, sus atributos y constructor son:

```
//Contenedor sobre el que se agrega este contenedor
protected ContenedorGenerico _contenedor;
//Imagen de fondo del contenedor
protected Imagen _imagenFondo;
//Elementos agregados al contenedor
protected List<Elemento> _elementos;
//Pantalla en la que se encuentra el contenedor
protected PantallaGenerica _pantalla;
//Coordenadas X,Y donde se ubica el contenedor
protected Vector2 _posicion;
//Espacio que se dejará entre los elementos agregados al contenedor
protected const int ESPACIO = 10;
//String identificadora del contenedor
protected string _id;
//Ancho del contenedor
protected int _ancho;
//Alto del contenedor
protected int _alto;
//Booleano que indica si se redimensionó el contenedor
protected bool _redimensionado;
//Booleano que indica si ya se ha cargado y se puede mostrar en
pantalla
protected bool _cargado;
//Booleano que indica si se va a pintar el contenedor en pantalla
protected bool _mostrar;
//Último elemento agregado al contenedor
protected Elemento ultimo = null;

public ContenedorGenerico(PantallaGenerica pantalla, Vector2 posicion,
ContenedorGenerico contenedor, string id)
{
    _pantalla = pantalla;
    _posicion = posicion;
    _contenedor = contenedor;
    _elementos = new List<Elemento>();
    _cargado = false;
    _redimensionado = false;
    _mostrar = true;
    _id = id;
}
```

En cuanto a la clase ContenedorSeleccionPersonaje los atributos y constructor son:

```
//Imagen de fondo del contenedor
private Imagen _fondo;
//Imagen del personaje que se mostrará
private Imagen _personaje;
//Nombre del personaje que se mostrará
private Texto _nombrePersonaje;
//Botón para eliminar un personaje
private Boton _eliminar;
//Botón para seleccionar un personaje
private Boton _seleccionar;
//Booleano que indica si la ranura está libre para la creación o ya
existe el personaje
private bool _creacion;
```

```
//Entero para controlar que no se accionen eventos de varios botones a
la vez
private int _updActual = -1;
//Constante usada en combinación con _updActual para controlar que no
se accionen eventos de varios botones a la vez
private const int _updLimite = 60;
//Elemento EntradaTexto para que el usuario seleccione el nombre con
que se creará el personaje
private EntradaTexto _texto;

public ContenedorSeleccionPersonaje(PantallaGenerica pantalla, Vector2
posicion, ContenedorGenerico contenedor, string nombre, string
textura, string id, bool creacion)
    : base(pantalla, posicion, contenedor, id)
{
    this._creacion = creacion;

    if (!creacion)
    {
        Vector2 maximo = new Vector2(380, 125);

        _fondo = new Imagen(this, new Vector2(), "fondo",
"SeleccionPJexistente", (int)maximo.X, (int)maximo.Y);

        _nombrePersonaje = new Texto(this, new Vector2(35, 8), nombre,
"Defecto", nombre, "izquierda", Color.White);

        _seleccionar = new Boton(this, new Vector2(),
"seleccionar_personaje", "botonSeleccionar");
        _seleccionar.AgregarObservador(this);
        _seleccionar.Posicion = new Vector2(160, maximo.Y * 0.45f -
(_seleccionar.ObtenerAlto() * 0.5f));

        _eliminar = new Boton(this, new Vector2(),
"eliminar_personaje", "botonEliminar");
        _eliminar.AgregarObservador(this);
        _eliminar.Posicion = new Vector2(160, _seleccionar.Posicion.Y
+ (_eliminar.ObtenerAlto() * 0.5f) + 20);

        _personaje = new Imagen(this, new Vector2(60,
_seleccionar.Posicion.Y + 10), "imagen_" + nombre, textura, 0, 0, 50,
50);
    }
    else
    {
        Vector2 maximo = new Vector2(380, 125);
        _fondo = new Imagen(this, new Vector2(), "fondo",
"SeleccionPJexistente", (int)maximo.X, (int)maximo.Y);

        _nombrePersonaje = new Texto(this, new Vector2(35, 8), "id_",
"Defecto", "- Ranura libre -", "izquierda", Color.White);
        _texto = new EntradaTexto(this, new Vector2((maximo.X * 0.5f)
- 125, (maximo.Y * 0.5f) - 20), 250, _nombrePersonaje.ObtenerAlto(),
            "id_et", "FondoGrisSeleccionado",
"FondoGrisNoSeleccionado", "Defecto", " >", 8, false);

        _seleccionar = new Boton(this, new Vector2(),
"crear_personaje", "botonCrear");
        _seleccionar.AgregarObservador(this);
```

```
        _seleccionar.Posicion = new Vector2(_texto.Posicion.X +
_texto.ObtenerAncho() - _seleccionar.ObtenerAncho(), _texto.Posicion.Y
+ +_texto.ObtenerAlto() + 10);
    }
}
```

Volviendo al aspecto de la comunicación, en estos contenedores se han definido como observadores para los eventos producidos en los botones de selección, eliminación y creación de personajes los mismos contenedores sobre los que están insertados los botones. De modo que la clase ContenedorSeleccionPJ tendrá el siguiente método Notificar():

```
public void Notificar(Observado o, int evento, object auxiliar)
{
    if (_updActual == -1)
    {
        switch (evento)
        {
            case 1:
            {//Se presiona un boton
                string id = (string)auxiliar;
                if (auxiliar.Equals("eliminar_personaje"))
                {
                    if
(_pantalla.Juego.Cliente.EliminarPersonajeSL(_nombrePersonaje.MiTexto)
)
                    {//Se ha borrado correctamente
                        this.ConvertirCreacion();
                    }
                }
                else if (auxiliar.Equals("seleccionar_personaje"))
                {
                    //Seleccionamos el personaje

_pantalla.Juego.Cliente.SeleccionarPersonajeSL(_nombrePersonaje.MiText
o);
                    //Desconectamos de SL
                    _pantalla.Juego.Cliente.DesconectarSL();

                    //Pasamos a la siguiente pantalla
                    _pantalla.Juego.PantallaSiguiente();
                }
                else if (auxiliar.Equals("crear_personaje"))
                {
                    if
(_pantalla.Juego.Cliente.CrearPersonajeSL(_texto.MiTexto))
                    {//Se ha creado correctamente
                        this.ConvertirSeleccion();
                    }
                }
                break;
            }
        }
    }
}
```

De este modo se permite al jugador:

- *Eliminar un personaje*

Se ejecutará el método EliminarPersonajeLS(…) de la clase InterfazCliente, que tendrá el siguiente código:

```
public bool EliminarPersonajeSL(string pj)
{
    Paquete pg = new Paquete();
    pg.AgregarElemento(_id_sl);
    pg.AgregarElemento(2); //2=borrar pj
    pg.AgregarElemento(pj);
    Paquete aux = _c.EnviarSL(pg);

    int respuesta = (int)aux.ObtenerElemento(0);
    if (respuesta == 0)
        return false;
    else
        return true;
}
```

Que a su vez llamará al método EnviarSL(…) de la clase Cliente, que lo que hace es enviar paquetes al servidor con el siguiente código:

```
public Paquete EnviarSL(Paquete pg)
{

    #region Comandos posibles
    // 1er Byte para el comando, 2o Byte para opciones adicionales
(nombre, pj, etc)
    // 0. Crear un personaje (Se acompaña del nombre del personaje)
    // 1. Seleccionar un personaje (Se acompaña de un segundo Byte con
el número del pj)
    // 2. Borrar un personaje (Se acompaña del número del pj)
    #endregion
    Paquete res = null;
    envioSL.Send(pg.Datos);

    byte[] datos = new byte[BYTES];
    envioSL.Receive(datos);
    res = new Paquete(datos);

    return res;
}
```

Obtendrá una respuesta del servidor en función de si se eliminó o no y convertirá el contenedor en uno de creación para que se pueda crear un nuevo personaje.

- *Seleccionar un personaje*

Se ejecutará el método SeleccionarPersonajeSL(…) de la clase InterfazCliente que realizará:

```
public bool SeleccionarPersonajeSL(string nombre)
{
    this._id_pj = _id_pj;
    _nombrePersonajeLocal = nombre;

    Paquete pg = new Paquete();
    pg.AgregarElemento(_id_sl);
    pg.AgregarElemento(1); //1=seleccionar pj
    pg.AgregarElemento(nombre);
    Paquete aux = _c.EnviarSL(pg);

    int respuesta = (int)aux.ObtenerElemento(0);
    if (respuesta == 0)
        return false;
    else
        return true;
}
```

Llamando al mismo método EnviarSL(…) de la clase Cliente expuesto anteriormente. Y desde el método Notificar() en la clase ContenedorSeleccionPJ se le indicará a la clase principal Juego que ya se ha seleccionado un personaje y que pase a la siguiente pantalla que será la de juego.

- *Crear un personaje*

Se ejecutará el método CrearPersonajeSL(…) de la clase InterfazCliente que realizará:

```
public bool CrearPersonajeSL(string nombre)
{
    Paquete pg = new Paquete();
    pg.AgregarElemento(_id_sl);
    pg.AgregarElemento(0); //0=crear pj
    pg.AgregarElemento(nombre);
    Paquete aux = _c.EnviarSL(pg);

    int respuesta = (int)aux.ObtenerElemento(0);
    if (respuesta == 0)
        return false;
    else
        return true;
}
```

Que llamará al mismo método EnviarSL(…) de la clase Cliente comentado en los apartados anteriores.

- **Negociación con el servidor de juego**

Una vez seleccionado un personaje, la clase principal Juego dará paso a la clase PantallaJuego tendrá los siguientes atributos y constructor:

```
//Lista de jugadores conectados
private List<Jugador> _jugadores;
//Instancia del jugador local
private Jugador _jugadorLocal;
//Jugador seleccionado para chat privado
private Jugador _jugadorSeleccionado;
//Contenedor del jugador seleccionado
private ContenedorGenerico _nombreJugadorSeleccionado;
//Instancia de cámara para el cálculo de coordenadas de los jugadores
private Camara _camara;
//Contador que establece cada cuanto se realizará un envío para
actualizar las coordenadas del jugador en el servidor
private const int CONTADOR = 8;
//Ticks que lleva el juego para combinar con CONTADOR y enviar
actualización al servidor
private int _ticks;
//Coordenadas del texto escrito en el chat
private Vector2 _posTextoChat;
//Atributo utilizado para ajustar el tiempo de carga de los gráficos
antes de su dibujado
private int ajuste;
//Instancia del mapa de juego
private Mapa _mapa;
//Hilo de recepción de información
private Thread _recepcion;
//Hilo de envío de información
private Thread _envio;

public PantallaJuego(Juego juego)
    : base (juego)
{
    _jugadores = new List<Jugador>();
    _camara = new Camara(this);

    _ticks = 0;

    //Conectamos con SPJ
    _juego.Cliente.ConectarSPJ();
    //Arrancamos el hilo que recibe datos del SPJ
    _recepcion = _juego.Cliente.RecibirSPJ();
    //Arrancamos el hilo que envia datos al SPJ
    _envio = _juego.Cliente.EnviarSPJ();
    //Enviamos el comando de conexión
    _juego.Cliente.EjecutarComandoSPJ(Cliente.CONEXION);
    //Establecemos el jugador clickado a NULL
    _jugadorSeleccionado = null;
    //Si no se ha seleccionado ningún jugador, pintamos el chat global
    _posTextoChat = new Vector2(7, _juego.Height - 168);
    _nombreJugadorSeleccionado = new ContenedorGenerico(this,
_posTextoChat, null, "nombreJugadorSeleccionado");
    _nombreJugadorSeleccionado.AgregarElemento(new
Texto(_nombreJugadorSeleccionado, new Vector2(), "texto", "defecto",
        "Chat global", "izquierda", Color.Black));

    //Introducimos la pantalla de chat
```

```
    ConstructorContenedor constructor = new
ConstructorContenedor(this);
    this.AgregarContenedor(constructor.CrearContenedorChat());
}
```

Como se puede ver en el código, durante la instanciación del objeto PantallaJuego se ejecuta el método ConectarSPJ() de la InterfazCliente que realizará:

```
public bool ConectarSPJ()
{
    this._id_spj = _c.ConectarSPJ();
    if (_id_spj == 0)
        return false;
    else
        return true;
}
```

Este a su vez llamará a ConectarSPJ() de la clase Cliente:

```
public int ConectarSPJ()
{
    #region Establecimiento de conexión dirección Cliente->Servidor.
Se guarda en envioSpj.
    IPHostEntry addressS = Dns.GetHostEntry(IP_SPJ);
    IPEndPoint ipepS = new IPEndPoint(Soporte.DevolverIPv4(addressS),
PUERTO_SPJ);

    envioSpj = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    envioSpj.NoDelay = true;

    try
    {
        envioSpj.Connect(ipepS);
    }
    catch (SocketException e)
    {
        return -1;
    }

    byte[] datos;

    Paquete pg = new Paquete();
      //Recibimos el identificador
    datos = new byte[BYTES];
    envioSpj.Receive(datos);
    pg = new Paquete(datos);
    this.id = (int)pg.ObtenerElemento(0);
    #endregion

    #region Establecimiento de conexión dirección Servidor->Cliente.
Se guarda en recepcionSpj.
    IPEndPoint ipepE = new IPEndPoint(IPAddress.Any,
puertoTcpRecepcion);
    Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    newsock.Bind(ipepE);
    newsock.Listen(10);
```

```
      newsock.Close();
    IPEndPoint clientepE = (IPEndPoint)recepcionSpj.RemoteEndPoint;
    #endregion

    //Recibo a los otros jugadores
    Paquete p;
    datos = new byte[BYTES];
    bool continuar = true;
    while (continuar)
    {
        recepcionSpj.Receive(datos);
        p = new Paquete(datos);

        if ((int)p.ObtenerElemento(0) == 1)
        {//Viene un jugador
            string nombre = (string)p.ObtenerElemento(1);
            int jx = (int)p.ObtenerElemento(2);
            int jy = (int)p.ObtenerElemento(3);
            int jid = (int)p.ObtenerElemento(4);
            int jmov = (int)p.ObtenerElemento(5);
            string textura = (string)p.ObtenerElemento(6);
            if (jid == id)
            {
                interfaz.CrearJugador(nombre, true, jx, jy, jid, jmov,
textura);
            }
            else
            {
                interfaz.CrearJugador(nombre, false, jx, jy, jid,
jmov, textura);
            }
        }
        else
        {
            continuar = false;
        }
        p = new Paquete();
        p.AgregarElemento(0); //Si quedan más elementos envíalos
        envioSpj.Send(p.Datos);
    }

    return this.id;
}
```

El código queda dividido en varias regiones, en la primera parte se establece conexión con el servidor de juego para enviarle información del jugador local y se recibe el identificador dentro de este servidor, en la segunda parte el servidor de juego establece conexión por otro socket con el jugador para que el jugador local reciba información de los jugadores remotos, finalmente se le comunica al jugador las coordenadas y movimientos de todos los jugadores actualmente conectados.

Una vez se ha conectado con el servidor de juego en ambas direcciones y se tiene la información de los jugadores, se asigna al hilo de recepción de datos el hilo devuelto por la ejecución del método RecibirSPJ de la clase InterfazCliente:

```
public Thread RecibirSPJ()
{
    Thread nuevo = new Thread(new ThreadStart(_c.Recibir));
    nuevo.Start();
    _juego.AgregarHilo(nuevo);
    return nuevo;
}
```

Que devuelve el hilo creado por el método Recibir() de la clase Cliente:

```
public void Recibir()
{
    byte[] data;
    while (!salir)
    {
        data = RecibirDeSPJ();
        if (!salir)
        {
            Paquete pg = new Paquete(data);
            interfaz.ActualizarJugador(pg);
        }
    }
}
```

Este último método envía paquetes de datos hasta que no se cumpla la condición de salida. Estos paquetes de datos los recoge a través del método RecibirDeSPJ() que está a la espera de paquetes desde el servidor de juego:

```
public byte[] RecibirDeSPJ()
{
    byte[] data = new byte[BYTES];
    try
    {
        recepcionSpj.Receive(data);
    }
    catch (Exception ex) { }
    return data;
}
```

En cuanto al hilo de envío, vendrá devuelto por el método EnviarSPJ() de la clase InterfazCliente:

```
public Thread EnviarSPJ()
{
    Thread nuevo = new Thread(new ThreadStart(_c.EnviarSPJ));
    nuevo.Start();
    _juego.AgregarHilo(nuevo);
    return nuevo;
}
```

Que de forma análoga al hilo de recepción usa el método EnviarSPJ() de la clase Cliente que realizará:

```
public void EnviarSPJ()
{
    while (!salir)
    {
        if (_comandos.Count > 0)
        {
            Paquete pg = new Paquete();
            lock (mutex)
            {
                pg.AgregarElemento(id);//Le enviamos el identificador
del cliente
                if (_comandos[0] != MENSAJECHAT)
                {
                    pg.AgregarElemento(_comandos[0]);//Le enviamos el
movimiento
                    _comandos.RemoveAt(0);
                    Jugador j = interfaz.Local;
                    Vector2 posicion = j.PosicionMovimiento;
                    pg.AgregarElemento((int)posicion.X);
                    pg.AgregarElemento((int)posicion.Y);
                }
                else
                {
                    pg.AgregarElemento(_comandos[0]);//Le enviamos el
movimiento
                    _comandos.RemoveAt(0);
                    pg.AgregarElemento(_mensajeschat[0]);
                    _mensajeschat.RemoveAt(0);
                }
            }

            envioSpj.Send(pg.Datos);
        }
    }
}
```

Que como peculiaridad bloqueará el acceso a la variable mutex como mecanismo de sincronización de acceso a variables compartidas, ya que sobre el array _comandos[] será donde se inserten los paquetes a enviar por parte del hilo de ejecución del juego, y que a su vez puede estar siendo accedido por este hilo de envío.

Finalmente, si en el array _comandos[] se encuentra algún paquete, será enviado con el método Send.

Por otro lado, cada vez que el jugador local realice un nuevo movimiento este será enviado al servidor. Para ello en el método Update() de la clase PantallaJuego se introducirá el siguiente código:

```
public override void Update()
{
      //Código para la detección del movimiento del jugador

      //Código para la detección de colisiones

      _juego.Cliente.EjecutarComandoSPJ(mov);

      //Código para el cálculo de posiciones de jugadores y mapas

      . . .

}
```

Que llamará al método EjecutarComandoSPJ(…) de la clase InterfazCliente:

```
public void EjecutarComandoSPJ(int comando)
{
    _c.EjecutarComandoSPJ(comando);
}
```

Que a su vez llamará al método EjecutarComandoSPJ(…) de la clase Cliente:

```
public void EjecutarComandoSPJ(int com)
{
    if (com == DESCONEXION)
    {
        this.DesconectarSPJ();
    }
    else
    {
        this.AgregarComando(com);
    }
}
```

Que en el caso de ser un comando de desconexión ejecutará el método DesconectarPJ():

```
public bool DesconectarSPJ()
{
    salir = true;
    AgregarComando(QUIETO);
    AgregarComando(DESCONEXION);
    try
    {
        envioSpj.Shutdown(SocketShutdown.Both);
        envioSpj.Close();
        recepcionSpj.Shutdown(SocketShutdown.Both);
        recepcionSpj.Close();
        return true;
    }
    catch (Exception ex)
    {
        return false;
```

```
    }
}
```

Y en el caso de cualquier otro comando ejecutará AgregarComando(…):

```
public void AgregarComando(int n)
{
    lock (mutex)
    {
        _comandos.Add(n);
    }
}
```

Que bloqueará la variable de sincronización mutex para evitar el envío y actualización de movimientos de forma simultánea.

5.2.8.Sistema de chat

El sistema de chat hará uso del ComponenteGenerico Chat explicado en el apartado 5.2.5. Eventos en la interacción del usuario y se encontrará empotrado en la pantalla de juego para permitir al jugador local comunicarse con los jugadores remotos como muestra la siguiente imagen:

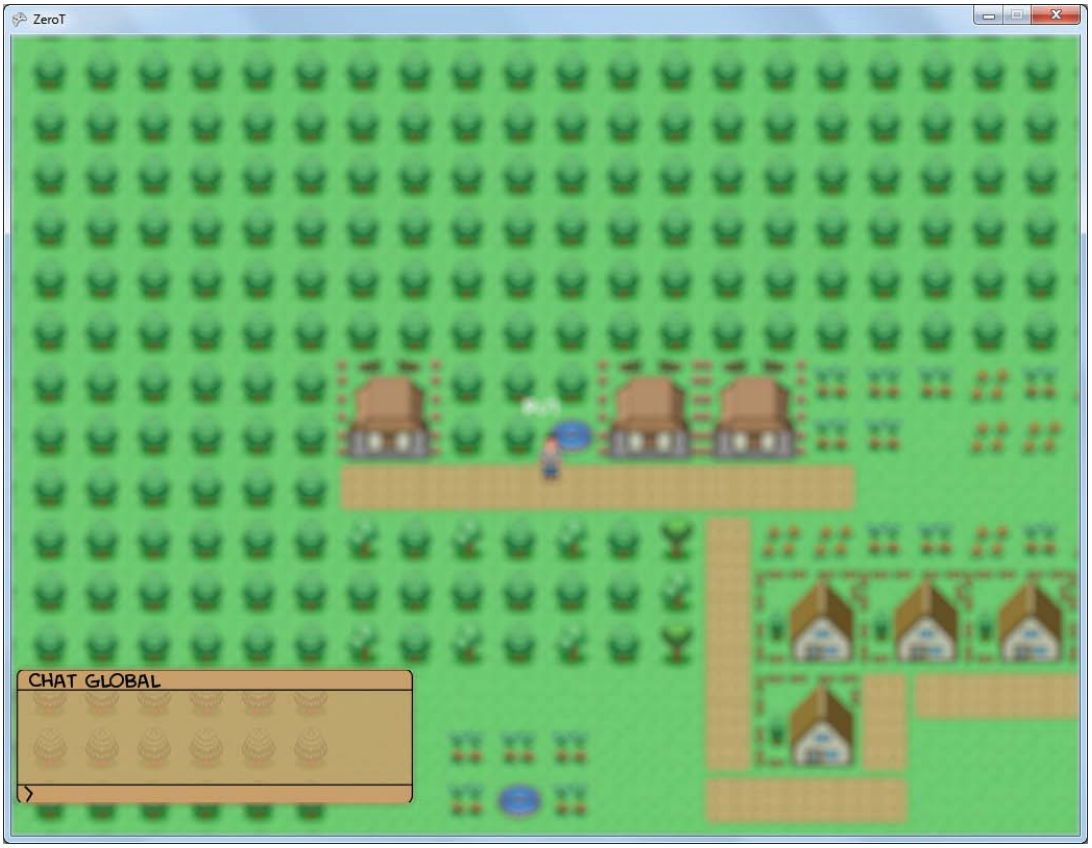

Imagen 48. Control para el sistema de chat

Como se puede observar, la interfaz del chat está dividida en tres secciones:

- **Receptores del mensaje**

La primera sección sirve para indicar al jugador a quien va a ir dirigido el mensaje, en el caso que se muestra en la captura lo leerán todos los jugadores conectados a la partida, pero también puede ir dirigida a un jugador específico, de modo que el resto de jugadores no lo leerán.

Por lo que si el jugador local quiere enviar un mensaje privado a otro jugador, tendrá que pinchar con el ratón sobre el jugador con el que se quiere comunicar.

El código para realizar esta acción estará introducido en el método Update() de la clase PantallaJuego y será el siguiente:

```
public override void Update()
{
      //Código para la detección del movimiento del jugador
      //Código para la detección de colisiones
    //Aquí controlamos si se clickea sobre un jugador
    if (_juego.RatonActual.LeftButton == ButtonState.Pressed)
    {
        Vector2 posicion = new Vector2(_juego.RatonActual.X,
_juego.RatonActual.Y);
        Jugador j = this.SeleccionarJugador(posicion);
        if (j != null)
        {
            _jugadorSeleccionado = j;
            //Se ha seleccionado un jugador, pintamos su nombre en el
chat
            _nombreJugadorSeleccionado = new ContenedorGenerico(this,
_posTextoChat, null, "nombreJugadorSeleccionado");
            _nombreJugadorSeleccionado.AgregarElemento(new
Texto(_nombreJugadorSeleccionado, new Vector2(), "texto", "defecto",
                "Chat local con " + _jugadorSeleccionado.Nombre,
"izquierda", Color.Black));
        }
    }

    //Si se presiona ESC se deselecciona el jugador
    if (_juego.TecladoActual.IsKeyDown(Keys.Escape))
    {
        _jugadorSeleccionado = null;
        //No se ha seleccionado un jugador, pintamos el chat global
        _nombreJugadorSeleccionado = new ContenedorGenerico(this,
_posTextoChat, null, "nombreJugadorSeleccionado");
        _nombreJugadorSeleccionado.AgregarElemento(new
Texto(_nombreJugadorSeleccionado, new Vector2(), "texto", "defecto",
            "Chat global", "izquierda", Color.Black));
    }

    if (_nombreJugadorSeleccionado != null)
    {
        _nombreJugadorSeleccionado.Update();
    }

      //Código para la ejecución de comandos y envío de acciones al
servidor
      //Código para el cálculo de posiciones de jugadores y mapas
      . . .
}
```

Como se puede ver en el código, se comprueba si se ha presionado sobre el botón izquierdo del ratón, y en ese caso se coge la coordenada donde se ha hecho click y se ejecuta el método SeleccionadJugador(…) sobre esa coordenada, el código de este método es el siguiente:

```
private Jugador SeleccionarJugador(Vector2 posicion)
{
    Rectangle mouse = new Rectangle((int)posicion.X, (int)posicion.Y,
2, 2);
    Rectangle aux;

    for (int i = 0; i < _jugadores.Count; i++)
    {
        aux = new Rectangle((int)_jugadores[i].Grafico.Posicion.X,
(int)_jugadores[i].Grafico.Posicion.Y, 50, 90);
        if (aux.Intersects(mouse))
        {
            return _jugadores[i];
        }
    }

    return null;
}
```

Utilizando la misma idea que en el caso del cálculo del sistema de colisiones, se define un rectángulo por cada jugador, y un rectángulo en la posición donde se hizo click, y se comprueba si se produce una intersección de estos, en cuyo caso se devuelve el jugador sobre el que se ha hecho click.

Una vez se tiene el nombre del jugador seleccionado, se introducirá en el campo de a quién van dirigido los mensajes (nombreJugadorSeleccionado) para informar al jugador local de con quién chateará en privado.

En el código introducido en el método Update() también se puede ver un fragmento de código que controla si se presionó la tecla Escape, la cual lo que hará es la función inversa a seleccionar a un personaje para chatear en privado, es decir, deseleccionará al jugador previamente seleccionado y permitirá volver a chatear con el resto de jugadores de forma global.

- **Mensajes escritos**

En la sección intermedia de la interfaz del chat se escribirán tantos los mensajes del jugador local como los mensajes del resto de jugadores, para ello estos mensajes tienen que llegar desde el servidor al hilo de recepción de datos del juego.

Como quedó explicado en el apartado anterior, este hilo recibirá mensajes a través del método RecibirDeSPJ() en la clase Cliente, creando un Paquete con los bytes recibidos en el método Recibir() de la clase Cliente y ejecutando el método ActualizarJugador(…) de la clase InterfazCliente para actualizar la información de los jugadores, en este caso, los mensajes de chat enviados, por lo que este método tendrá el siguiente código:

```
public void ActualizarJugador(Paquete pg)
{
    int id_jugador = (int)pg.ObtenerElemento(0);
    int comando = (int)pg.ObtenerElemento(1);
    int x = -1;
    int y = -1;
```

```
    try
    {
        x = (int)pg.ObtenerElemento(2);
        y = (int)pg.ObtenerElemento(3);
    }
    catch
    {
        //Si llega aquí se trata de una desconexión porque ese paquete
no tiene Vector2s
    }
    if(comando == Cliente.MENSAJECHAT)
    {
        //Aquí entra un mensaje de chat
        string mensaje = (string)pg.ObtenerElemento(2);
        //Actualizamos la ventana de chat
 ((PantallaJuego)_juego.Pantalla).Chat.ActualizarTextos(mensaje);
    }
}
```

Como se puede ver, si el paquete enviado tiene en el campo comando codificado un byte indicando que se trata de un mensaje de chat, la InterfazCliente ejecutará el método ActualizarTextos(…) de la clase Chat con el nuevo mensaje, este método tendrá la siguiente codificación:

```
public void ActualizarTextos(string nuevoMsg)
{
    //Comprobamos si el mensaje es para este usuario
    int punto = nuevoMsg.IndexOf('.');
    string destino = nuevoMsg.Substring(0, punto);
    string mensaje = nuevoMsg.Substring(punto+1);
    if (destino.Equals("TODOS") ||
destino.Equals(_pantallaJuego.JugadorLocal.Nombre) ||
mensaje.Contains(_pantallaJuego.JugadorLocal.Nombre+":"))
    {
        _textos[0].MiTexto = _textos[1].MiTexto;
        _textos[1].MiTexto = _textos[2].MiTexto;
        _textos[2].MiTexto = _textos[3].MiTexto;
        _textos[3].MiTexto = mensaje;
    }
}
```

En este código se puede ver como navega la información de los mensajes de chat, la cual tiene la siguiente estructura:

<NOMBRE DEL PERSONAJE EMISOR>.<MENSAJE DE TEXTO>

De este modo lo que el código hará será dividir el mensaje en dos partes, el destino del mensaje que puede ser a todos a un único personaje, y el mensaje que se envía, y en caso de que el mensaje se envíe a todos o al personaje del jugador local, se hará un movimiento de mensajes, ya que le chat sólo muestra 4 mensajes a la vez ordenados de arriba abajo por orden de llegada.

Finalmente el método de representación gráfica Draw() de la clase Chat se encargará de dibujar estos textos en pantalla:

```
public override void Draw()
{
    for (int i = 0; i < _textos.Length; i++)
    {
        _textos[i].Draw();
    }
    _entrada.Draw();
}
```

- **Entrada de texto del jugador local**

La tercera y última sección será un ComponenteGenerico simple de tipo EntradaTexto empotrado en el ComponenteGenerico compuesto Chat.

En él, una vez que el jugador local haya escrito y presionado la tecla Enter y debido al código introducido en el constructor del Chat como se vio en el apartado 5.2.5. Eventos en la interacción del usuario, la EntradaTexto notificará a su observador Chat de que se ha presionado Enter habiendo introducido un mensaje.

De modo que al método Notificar() de Chat llegará el mensaje escrito en la EntradaTexto, y el código que realizará este método Notificar() será:

```
public override void Notificar(Observado o, int evento, object
auxiliar)
{
    for (int i = 0; i < _observadores.Count; i++)
    {
        _observadores[i].Notificar(this, 24, auxiliar);
    }

    //Vaciamos la entrada de texto
    _entrada.MiTexto = "";
}
```

Es decir, la EntradaTexto notifica al Chat el mensaje que se ha escrito en él, y el Chat notifica a sus observadores (PantallaJuego) ese mensaje que le ha llegado a él con un código de evento 24. Además borrará el texto escrito por el jugador en la entrada de texto para que sepa que el mensaje ya se envió.

Por lo que la PantallaJuego tendrá que controlar esta notificación en el método Notificar de su clase, que tendrá la siguiente codificación:

```
public override void Notificar(ZeroT.Patrones.Observer.Observado o,
int evento, object auxiliar)
{
      if (evento == 24)
    {
        //Enviamos al hilo cliente el mensaje para que lo reciban
todos los usuarios
        string s = ((string)auxiliar);
        if (s[s.Length - 1] == '|')
        {
```

```
            s = s.Substring(2, s.Length - 3);
        }
        else
        {
            s = s.Substring(2, s.Length - 2);
        }
        _juego.Cliente.EnviarMensajeChat(s);
    }
}
```

Este método eliminará el carácter | introducido en el parpadeo de la EntradaTexto en caso de que lo haya, y ejecutará el método EnviarMensajeChat de la InterfazCliente, que realizará:

```
public void EnviarMensajeChat(string mensaje)
{
    PantallaJuego pantalla = (PantallaJuego)_juego.Pantalla;
    if (pantalla.JugadorSeleccionado != null)
    {
        _c.EnviarMensajeChat(pantalla.JugadorSeleccionado.Nombre + "."
+ _nombrePersonajeLocal + ": " + mensaje);
    }
    else
    {
        _c.EnviarMensajeChat("TODOS."+ _nombrePersonajeLocal + ":" +
mensaje);
    }
}
```

Es decir, cogerá el mensaje y construirá la estructura de mensajería expuesta cuando se trató la recepción de los mensajes en la sección 2 del Chat:

<NOMBRE DEL PERSONAJE EMISOR>.<MENSAJE DE TEXTO>

Cuando se tiene construido este mensaje se envía haciendo uso del método EnviarMensajeChat(…) de la clase Cliente:

```
public void EnviarMensajeChat(string mensaje)
{
    _mensajeschat.Add(mensaje);
    this.AgregarComando(MENSAJECHAT);
}
```

El cual como se puede ver agrega el comando MENSAJECHAT en el array _comandos a través del método AgregarComando(…) para que el hilo de envío de paquetes lo envíe cuando proceda.

5.2.9.Recepción de la información de otros jugadores

Como se comentó en el apartado anterior, el hilo de recepción de datos desde el servidor recibirá paquetes a través del método RecibirDeSPJ() en la clase Cliente, creando un Paquete con los bytes recibidos en el método Recibir() de la clase Cliente y ejecutando el método ActualizarJugador(…) de la clase InterfazCliente para actualizar la información de todos los jugadores, teniendo el siguiente código:

```
public void ActualizarJugador(Paquete pg)
{
    int id_jugador = (int)pg.ObtenerElemento(0);
    int comando = (int)pg.ObtenerElemento(1);
    int x = -1;
    int y = -1;

    try
    {
        x = (int)pg.ObtenerElemento(2);
        y = (int)pg.ObtenerElemento(3);
    }
    catch
    {
        //Si llega aquí se trata de una desconexión porque ese paquete
no tiene Vector2s
    }

    if (comando == Cliente.CONEXION && id_jugador != _local.Id)
    {
        //Conexión de un jugador externo
        string nombre = (string)pg.ObtenerElemento(4);
        int movimiento = (int)pg.ObtenerElemento(5);
        string textura = (string)pg.ObtenerElemento(6);
        this.CrearJugador(nombre, false, x, y, id_jugador, movimiento,
textura);
    }
    else if (comando == Cliente.DESCONEXION && id_jugador !=
_local.Id)
    {
        //Desconexión de un jugador externo
        this.EliminarJugador(id_jugador);
    }
    else if(comando == Cliente.MENSAJECHAT)
    {
        //Aquí entra un mensaje de chat
        string mensaje = (string)pg.ObtenerElemento(2);
        //Actualizamos la ventana de chat
 ((PantallaJuego)_juego.Pantalla).Chat.ActualizarTextos(mensaje);
    }
    else if (id_jugador == _local.Id)
    {
        //Comando del jugador de esta maquina

    }
    else if (id_jugador != _local.Id)
    {
        //Comando de otro jugador
        //_localizamos al jugador
        Jugador aux = null;
        for (int i = 0; i < _jugadores.Count; i++)
```

```
        {
            if (_jugadores[i].Id == id_jugador)
            {
                aux = _jugadores[i];
                i = _jugadores.Count;
            }
        }
        lock (aux)
        {
            aux.FinalizarMovimiento(comando, new Vector2(x, y));
        }

    }
}
```

Como se puede ver en el código, se obtiene del Paquete el identificador del jugador, el comando que está ejecutando (movimiento y de qué tipo, si es un mensaje de chat, etc.), y además se intenta recuperar las coordenadas donde ejecutó el comando, ya que en caso de que no venga esta información con el paquete se tratará de una desconexión del cliente.

Una vez se tienen estos datos, en función del comando se realizará una acción:

- **Conexión de un nuevo jugador a la partida**

En la conexión de un nuevo jugador a la partida, la InterfazCliente se obtendrá información adicional del jugador como el nombre de éste, el movimiento que está realizando en la conexión (campo importante en caso de que se trate de que el jugador local se conecta y uno de los jugadores se está moviendo en ese momento, tendrá que recibir la información de que ese jugador que ya estaba jugando no está quieto) y la textura que utilizará la imagen del personaje nuevo.

Con todos estos datos se ejecutará el método CrearJugador(…) que tendrá el siguiente código:

```
public Jugador CrearJugador(string nombre, bool _local, int x, int y,
int id, int movimiento, string textura)
{
    Jugador j = new Jugador(nombre, _local, id, x, y, _juego.Pantalla,
movimiento, textura);
    if (_local)
    {
        this._local = j;
        this.NotificarObservadores(23, this._local);
    }

    //Si ya existe el jugador, no le añadimos
    for (int i = 0; i < _jugadores.Count; i++)
    {
        if (_jugadores[i].Id == id)
            return null;
    }

    _jugadores.Add(j);
    return j;
}
```

Como se puede ver en el código, se creará el nuevo jugador, en caso de que se trate del jugador local se notificará a los observadores (en este caso PantallaJuego) para que comience la carga del mapa para poder representar a los jugadores, y finalmente se agregará al jugador a la lista de jugadores conectados a la partida.

La notificación a PantallaJuego supondrá controlar el evento con identificado número 23, de modo que el método Notificar() de la clase PantallaJuego quedará finalmente:

```
public override void Notificar(ZeroT.Patrones.Observer.Observado o,
int evento, object auxiliar)
{
    if (evento == 23)
    {
        //Se lanza cuando el SPJ devuelve el jugador local al cliente.
        _jugadorLocal = (Jugador)auxiliar;

        _mapa = new Mapa("mapa.txt", "mapeado", this);
    }
    else if (evento == 24)
    {
        //Enviamos al hilo cliente el mensaje para que lo reciban
todos los usuarios
        string s = ((string)auxiliar);
        if (s[s.Length - 1] == '|')
        {
            s = s.Substring(2, s.Length - 3);
        }
        else
        {
            s = s.Substring(2, s.Length - 2);
        }
        _juego.Cliente.EnviarMensajeChat(s);
    }
}
```

- **Desconexión de un jugador de la partida**

En el caso de la desconexión de un jugador de la partida, se ejecutará el método EliminarJugador(…) de la clase InterfazCliente que tiene la siguiente codificación:

```
public void EliminarJugador(int id)
{
    for (int i = 0; i < _jugadores.Count; i++)
    {
        if (_jugadores[i].Id == id)
        {
            _jugadores.RemoveAt(i);
            i = _jugadores.Count;
        }
    }
}
```

Recorrerá el array de jugadores en busca del jugador con el identificador recibido en el paquete y eliminará para no seguir actualizando sus datos ni dibujar en pantalla

- **Recepción de un mensaje de chat**

Esta funcionalidad quedó explicada en el apartado anterior 5.2.8. Sistema de chat.

- **Otro tipo de comando**

Si llega a esta sección de código, entonces lo que se está recibiendo es un comando de movimiento en alguna dirección por parte del jugador, de modo que se localiza al jugador en el array de jugadores, se bloquea el acceso a los datos de este jugador como mecanismo de sincronización para asegurar que no se acceden a sus atributos mientas están siendo actualizados, y se ejecuta el método FinalizarMovimiento(…) sobre el Jugador, que realizará:

```
public void FinalizarMovimiento(int nuevoMovimiento, Vector2 posicion)
{
    this.movimiento = nuevoMovimiento;
    this.instante = DateTime.Now;
    this.posicionAbsoluta = new Vector2(posicion.X, posicion.Y);
}
```

Este código rectificará la posición del jugador en caso de que se haya estado calculando mal en la pantalla de juego debido a problemas de latencia, establecerá el instante de tiempo en que se recibió el último paquete de ese jugador y la dirección del movimiento que está realizando para que el juego pueda seguir calculando sus coordenadas hasta que se reciba el siguiente paquete por su parte.

5.3.Aplicación Servidor de Conexión

La aplicación Servidor de Conexión se ejecutará en el lado del servidor. Será desarrollada en lenguaje C# como una aplicación de consola ya que no se considera necesario que ningún usuario interactúe con ella a través de una interfaz gráfica.

Será la encargada de dirigir las conexiones establecidas por el resto de servidores y de clientes, siendo capaz de conocer en todo momento qué servidores y de qué tipo son los que están activos en el sistema para comunicarlos a los clientes cuando inicien una partida.

5.3.1.Proceso de conexión y desconexión de servidores

La aplicación Servidor de Conexión tendrá una clase ServidorSC que será la encargada de manejar las conexiones recibidas y gestionar la información que pase por ella, para ello contará con los siguientes atributos y constructor:

```
//Dirección del host
private String host;
//Puertos de conexión para los socket
private int puertoTcpRecepcion;
//Nombre del fichero de configuración
private const String FICHERO  = "./configuracion.txt";
//Lista con los servidores de juego
static List<Servidor> SPJ;
//Servidor de login
static List<Servidor> SL;
//TAMAÑO DE ARRAY DE BYTES
private const int BYTES = 1024;

public ServidorSC()
{
    SPJ = new List<Servidor>();
    SL = new List<Servidor>();
    host = ObtenerDatoDeFichero("host");
    puertoTcpRecepcion = Int32.Parse(ObtenerDatoDeFichero("tcp_r"));

    mostrar = true;
}
```

Por otro lado el método Main(…) tendrá la siguiente codificación:

```
static void Main(string[] args)
{
    InterfazServidorSC interfaz = new InterfazServidorSC();
    interfaz.ArrancarServidor();
}
```

Que llamará al método ArrancarServidor() con el código:

```
public void ArrancarServidor()
{
    ServidorSC sc = new ServidorSC();
    sc.RecibirConexionesTCP();
}
```

De vuelta al proceso de instanciación del ServidorSC, se hace uso del método ObtenerDatoDeFichero(), cuyo código es el siguiente:

```
private String ObtenerDatoDeFichero(String dato)
{
    // Definimos un StreamReader
    StreamReader lector;
    String linea = "";
    String res = "";

    try
    {
        lector = File.OpenText(FICHERO);

        while ((linea = lector.ReadLine()) != null)
        {
            if (!linea.Contains("#") && !linea.Equals(""))
            {
                char[] delim = new char[1];
                delim[0] = ':';
                String[] tokens = linea.Split(delim,
StringSplitOptions.RemoveEmptyEntries);
                if (tokens[0].Equals(dato))
                {
                    res = tokens[1].Trim();
                    break;
                }
            }
        }
        lector.Close();
    }
    catch (Exception e)
    {
        return null;
    }
    return res;
}
```

Como se puede ver, accede al fichero de configuración y recupera el valor del campo recibido como parámetro, ya que en el fichero de configuración la información constará con el siguiente formato:

<clave>: <valor>
<clave>: <valor>
…

Una vez se haya instanciado un objeto de tipo ServidorSC, se llamará a su método RecibirConexionesTCP() el cual gestionará todas las conexiones entrantes de los servidores y las almacenará en sus estructuras de datos para comunicárselas posteriormente a los jugadores que inicien una nueva partida. El código base de este método RecibirConexionesTCP() será:

```
public void RecibirConexionesTCP()
{
    byte[] data;
```

```
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
puertoTcpRecepcion);
    Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    newsock.Bind(ipep);
    newsock.Listen(10);
    while (true)
    {
        data = new byte[BYTES];
        Socket client = newsock.Accept();
        client.NoDelay = true;
        IPEndPoint clientep = (IPEndPoint)client.RemoteEndPoint;
        client.Receive(data);
        Paquete generico = new Paquete(data);
        int comando = (int)generico.ObtenerElemento(0);
//Aquí habrá que gestionar la conexión según el comando recibido
        client.Close();
    }
    newsock.Close();
}
```

Como se puede ver en el código, se crea un socket que queda a la espera de conexiones entrantes por el puerto leído del fichero de configuración y una vez establecida esta conexión recibe un Paquete como el que se explicó en el apartado 5.2.7. Envío de la información del jugador.

De este paquete se recuperará el primer elemento el cual indicará el comando recibido, y en función de este comando se sabrá si está realizando una conexión un servidor y qué tipo de conexión, o si se trata de un cliente.

Este código se introducirá con una instrucción **switch** en la línea de comentario "*Aquí habrá que gestionar la conexión según el comando recibido*".

Para el caso de la conexión y desconexión de servidores dependerá del tipo de servidor que inicie la conexión, teniendo cuatro casos posibles:

- **Conexión de un Servidor de Login**

En el caso de que sea un servidor de Login el que establece la conexión, se tendrá el siguiente código:

```
switch (comando)
{
    #region Conexión de un servidor SL
    case 2:
        {
            //Guardamos la dirección
            String sl_ip = clientep.Address.ToString();

            //Guardamos el puerto de escucha
            int sl_puerto = (int)generico.ObtenerElemento(1);

            //Le decimos que se ha registrado correctamente (1)
            Paquete pg = new Paquete();
            pg.AgregarElemento(1);
            client.Send(pg.Datos);
```

```
                SL.Add(new Servidor(sl_ip, sl_puerto));
                break;
            }
        #endregion
}
```

Como se puede ver en el código, si tras recibir una conexión en el Servidor de Conexión y el comando recibido en el paquete de conexión es un 2, entonces se trata de una conexión de un Servidor de Login, de modo que se guardará la dirección IP del servidor que está realizando la conexión.

Al tratarse de un paquete de conexión enviado al Servidor de Conexión con comando 2 y según se definió en el apartado 4.2.4. Protocolo de transporte, flujo de datos entre servidores y cliente, se sabe que este paquete tiene también un campo que indica el puerto por el que el servidor que se está dando de alta va a ofrecer servicio, por lo que se recupera esta información del paquete, y se añade a la lista de Servidores de Login dados de alta en el Servidor de Conexión un nuevo servidor.

Los atributos y constructor de la clase Servidor que encapsula la información de los servidores son los siguientes:

```
private String ip;
private int puerto;
public Servidor(String ip, int puerto)
{
    this.ip = ip;
    this.puerto = puerto;
}
```

Adicionalmente la clase tendrá sendas propiedades para la recuperación de estos datos:

```
public String IP
{
    get
    {
        return ip;
    }
}

public int Puerto
{
    get
    {
        return puerto;
    }
}
```

- **Desconexión de un Servidor de Login**

En el caso de que un Servidor de Login envíe un paquete de desconexión se tendrá el siguiente código dentro de la instrucción **switch**:

```
case 4:
    {
        DesconectarSL(clientep.Address.ToString());
        break;
    }
```

Como se puede ver el comando recibido para la desconexión de un Servidor de Login es 4, y en este caso no es necesario comunicar más información ya que a través de la dirección IP que envía este comando se podrá identificar el Servidor de Login a dar de baja. Este código hace referencia al siguiente método DesconectarSL(…):

```
private void DesconectarSL(String direccion)
{
    for (int i = 0; i < SL.Count; i++)
    {
        if (SL[i].IP.Equals(direccion))
        {
            SL.RemoveAt(i);
            i = SL.Count;
        }
    }
}
```

El cual localiza el Servidor de Login de su lista de servidores dados de alta a través de la dirección IP y lo elimina.

- **Conexión de un Servidor de Juego**

En el caso de que un Servidor de Juego envíe un paquete de conexión se tendrá el siguiente código dentro de la instrucción **switch**:

```
case 3:
    {
        //Guardamos la dirección
        String spj_ip = clientep.Address.ToString();

        //Guardamos el puerto de escucha
        int spj_puerto = (int)generico.ObtenerElemento(1);

        bool ok = true;
        //Informamos a los SL de que se ha conectado un SPJ
        for (int i = 0; i < SL.Count; i++)
        {
            Paquete pg_aux = new Paquete();
            pg_aux.AgregarElemento(1); //Comando de envio de SPJ
            pg_aux.AgregarElemento(spj_ip); //Ip del SPJ
            pg_aux.AgregarElemento(spj_puerto); //Puerto del SPJ

            pg_aux = Soporte.EnviarPaquete(SL[i].IP, SL[i].Puerto,
pg_aux);
```

```
            int respuesta = (int)pg_aux.ObtenerElemento(0);
            if (respuesta == 0) //Fallo
            {
                ok = false;
            }
        }

        //Informamos a SPJ de si se ha registrado o no
        Paquete pg = new Paquete();
        if (ok)
            pg.AgregarElemento(1);
        else
            pg.AgregarElemento(0);
        client.Send(pg.Datos);
        SPJ.Add(new Servidor(spj_ip, spj_puerto));
        break;
    }
```

Como se puede ver en el código, el comando recibido para la conexión del Servidor de Juego tiene valor 3, y al igual que el comando de conexión del Servidor de Login viene con un campo adicional que indica el puerto a través del cual el Servidor de Juego ofrecerá servicio.

Recibida la información de conexión del Servidor de Juego se informará a los Servidores de Login para que también la tengan, y finalmente se devolverá la respuesta de los Servidores de Login al Servidor de Juego que se está registrando, para ello se hará uso del método EnviarPaquete(…) de la clase Soporte, el cual tendrá la siguiente codificación:

```
public static Paquete EnviarPaquete(String direccion, int puerto,
Paquete pg)
{
    IPHostEntry address = Dns.GetHostEntry(direccion);
    IPEndPoint ipep = new IPEndPoint(DevolverIPv4(address), puerto);

    Socket server = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    try
    {
        server.Connect(ipep);
    }
    catch (SocketException e)
    {
        Paquete aux = new Paquete();
        aux.AgregarElemento(0);
        return aux;
    }

    server.Send(pg.Datos);

    byte[] data = new byte[1024];
    server.Receive(data);
    pg = new Paquete(data);

    server.Shutdown(SocketShutdown.Both);
    server.Close();

    return pg;
}
```

Como se puede observar en este código, se reenvía el paquete recibido del Servidor de Juego al Servidor de Login, y se devuelve el paquete con el que responde el Servidor de Login para que el Servidor de Conexión compruebe si la conexión fue satisfactoria e informar finalmente al Servidor de Juego.

- **Desconexión de un Servidor de Juego**

En el caso de que un Servidor de Juego envíe un paquete de desconexión se tendrá el siguiente código en el switch:

```
case 5:
    {
        DesconectarSPJ(clientep.Address.ToString());
        break;
    }
```

El código enviado por el Servidor de Juego tiene valor 5 y para darlo de baja en el Servidor de Conexión se ejecutará el método DesconectarSPJ(…) que tiene la siguiente codificación:

```
private void DesconectarSPJ(String direccion)
{
    for (int i = 0; i < SPJ.Count; i++)
    {
        if (SPJ[i].IP.Equals(direccion))
        {
            SPJ.RemoveAt(i);
            i = SPJ.Count;
        }
    }
}
```

Es decir, localizará al Servidor de Juego a través de la IP que solicita la desconexión y lo eliminará de la lista de Servidores de Juego registrados.

5.3.2.Proceso de conexión de clientes

El caso de los clientes difiere de los servidores de login y juego ya que el Servidor de Conexión no guardará una lista de los jugadores que están conectados ni tampoco tendrá que negociar con ellos la desconexión debido a que no estarán registrados en éste.

Para controlar la conexión de los clientes, al igual que en los comandos de conexión y desconexión de los distintos servidores, se introducirá el siguiente código en el switch de comandos recibidos en el Servidor de Conexión:

```
case 1:
    {
        Servidor sl = null;
        Servidor spj = null;
        if (SPJ.Count == 0 || SL.Count == 0)
        {
            Paquete pg = new Paquete();
            pg.AgregarElemento(0); //No se permiten conexiones porque
no hay SL o SPJ
            res = pg.Datos;
        }
        else
        {
            sl = ObtenerSL();
            spj = ObtenerSPJ();

            //Obtenemos los datos de SL
            byte[] aux = Soporte.EnviarComando(sl.IP, sl.Puerto, 0);
//0=Peticion de puerto
            if (aux == null)
                break;
            Paquete pg = new Paquete(aux);
            int puerto_sl = (int)pg.ObtenerElemento(1);
            //Obtenemos los datos de SPJ
            aux = Soporte.EnviarComando(spj.IP, spj.Puerto, 0);
//0=Peticion de puerto
            if (aux == null)
                break;
            pg = new Paquete(aux);
            int puerto_spj = (int)pg.ObtenerElemento(1);
            //Componemos el paquete del cliente
            pg = new Paquete();
            pg.AgregarElemento(1); //1=Confirmamos que todo ha ido
correctamente
            pg.AgregarElemento(sl.IP);
            pg.AgregarElemento(puerto_sl);
            pg.AgregarElemento(spj.IP);
            pg.AgregarElemento(puerto_spj);
            res = pg.Datos;
        }
        client.Send(res);
        break;
    }
```

En este código se puede ver el procedimiento a seguir en la conexión de los jugadores, recibiéndose en primer lugar un paquete de conexión cuyo comando es 1, en este punto se comprueba si hay disponibles servidores de juego y login, y de no haberlos se le devolverá al

jugador un paquete con valor 0 indicándole que no están registrados los servidores necesarios para jugar; en caso de estar registrados al menos un servidor de juego y login, continuará el procedimiento de conexión del jugador.

El siguiente paso es obtener un Servidor de Login y un Servidor de Juego para que el jugador comience la partida, para ello se llama a los siguientes dos métodos:

```
private Servidor ObtenerSL()
{
    return SL[0];
}

private Servidor ObtenerSPJ()
{
    return SPJ[0];
}
```

En el código se puede ver que siempre devuelven el primer servidor de cada tipo registrado en el Servidor de Conexión, este código se podría haber puesto directamente en el método RecibirConexionesTCP() pero se ha separado con la intención de realizar una mejor estructura que proporcione en el futuro la posibilidad del balanceo de carga en servidores saturados.

Una vez se han seleccionado los servidores que atenderán al jugador, se realiza una petición a estos para que habiliten un puerto al que se conectará el jugador llamando al método EnviarComando(…):

```
public static byte[] EnviarComando(String direccion, int puerto, int
comando)
{
    IPHostEntry address = Dns.GetHostEntry(direccion);
    IPEndPoint ipep = new IPEndPoint(DevolverIPv4(address), puerto);

    Socket server = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    try
    {
        server.Connect(ipep);
    }
    catch (SocketException e)
    {
        Console.WriteLine("Error en " + e.ToString());
    }

    Paquete pg = new Paquete();
    pg.AgregarElemento(comando);

    server.Send(pg.Datos);

    byte[] data = new byte[1024];
    int recv = server.Receive(data);

    server.Shutdown(SocketShutdown.Both);
    server.Close();
```

```
    return data;
}
```

Este método genera un paquete que enviará al servidor de login y de juego, y una vez estos hayan creado un nuevo hilo de ejecución para atender al jugador y hayan habilitado un nuevo socket por el que ofrecer servicio al jugador, devolverán un paquete con el puerto desde el que le atenderán.

Con la información de que servidor de login y juego van a atender al jugador, y los puertos por donde le ofrecerán servicio, el Servidor de Conexión enviará un paquete al jugador con los cinco datos necesarios:

- Un valor 1 para indicar que el juego está preparado para atenderle
- La dirección IP del Servidor de Login que le atenderá
- El puerto por el que le atenderá
- La dirección IP del Servidor de Juego que le atenderá
- El puerto por el que le atenderá

Enviados esos datos al jugador se cierra la conexión con éste para que pueda comenzar la partida.

5.4.Aplicación Servidor de Login

La aplicación Servidor de Login también se ejecutará en el lado del servidor, pero no será condición indispensable que se ejecute en la misma máquina que el Servidor de Conexión, ya que será el Servidor de Conexión el encargado de conocer las direcciones IP en las que los distintos servidores están siendo ejecutados. Será desarrollada en lenguaje C# como una aplicación de consola ya que no se considera necesario que ningún usuario interactúe con ella a través de una interfaz gráfica.

Será la encargada de dirigir el proceso de login de los clientes que se conecten al sistema, comprobando que los datos de usuario y cliente están dados de alta en el sistema, recuperando los personajes del jugador y permitiendo crear, eliminar y seleccionar el personaje con el que se comenzará la partida.

5.4.1.Proceso de alta en el sistema de servidores

El proceso de alta del Servidor de Login consiste en realizar una petición de registro al Servidor de Conexión el cual le indicará si todo fue correcto según se expone en el apartado 4.2.4. Protocolo de transporte, flujo de datos entre servidores y clientes.

Para ello se tendrá una clase ServidorLogin con los siguientes atributos y constructor:

```
//Dirección del host
private String host;
//Puerto de conexión TCP por el que recibe la información del Servidor
de Conexión
private int puertoTcpRecepcion;
//Puerto de conexión TCP por el que se conecta al Servidor de Conexión
private int puertoTcpEnvioSC;
//Nombre del fichero de configuración
static string FICHERO = "./configuracion.txt";
//Nombre del fichero con la base de datos de jugadores
static string FICHEROUSUARIOS = "./DBjugadores.txt";
//Conexiones de jugadores entrantes
static List<Jugador> entrada;
//Identificadores
static byte[] ids;
//Variable para mecanismo de sincronización en el acceso a variables
compartidas
static object mutex;
//TAMAÑO DE ARRAY DE BYTES
private const int BYTES = 1024;
//TAMAÑO MAXIMO DE JUGADORES SOPORTADOS
private const int JUGADORES = 2048;
//Lista de servidores SPJ
static List<ServidorSPJ> _listaSPJ = null;
private string _spjCliente;
//Control de finalización de hilos de ejecución
private bool salir;
```

```
public ServidorSL()
{
    entrada = new List<Jugador>();
    mutex = new object();
    ids = new byte[JUGADORES]; //2048 Jugadores

    host = ObtenerDatoDeFichero("host");
    puertoTcpRecepcion = Int32.Parse(ObtenerDatoDeFichero("tcp_r"));
    puertoTcpEnvioSC = Int32.Parse(ObtenerDatoDeFichero("tcp_s_sc"));

    mostrar = true;
}
```

Al igual que en el caso del Servidor de Conexión, para obtener los puertos por donde se atenderá y se ofrece servicio al Servidor de Login, así como el host desde donde se está ejecutando para poner el socket en funcionamiento, se cargarán los datos desde un fichero de configuración al que se accederá con el método ObtenerDatoDeFichero(…) y que tendrá la misma configuración que se expuso en el apartado 5.3.1. Proceso de conexión y desconexión de servidores.

Como método Main(…) se tendrá:

```
static void Main(string[] args)
{
    sl = new ServidorSL();

    if (sl.RegistrarEnSC())
    {
        sl.AtenderComandos();
    }
}
```

En este código se puede ver que el primer paso que realiza el Servidor de Login es registrarse en el Servidor de Conexión, esto lo hace a través del método RegistrarEnSC() que tiene la siguiente codificación:

```
public bool RegistrarEnSC()
{
    byte[] data = new byte[BYTES];
    IPHostEntry address = Dns.GetHostEntry(host);
    IPEndPoint ipep = new IPEndPoint(Soporte.DevolverIPv4(address),
puertoTcpEnvioSC);

    Socket server = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    server.NoDelay = true;

    try
    {
        server.Connect(ipep);
    }
    catch (SocketException e)
    {
        return false;
    }

    //Conexión
```

```
    //Mandamos el comando 2--> Conexión de SL en SC
    Paquete pg = new Paquete();
    pg.AgregarElemento(2); //Comando de conexión de SL en SC
    pg.AgregarElemento(puertoTcpRecepcion); //Puerto de recepción de
SL
    server.Send(pg.Datos);

    data = new byte[BYTES];

    int recv = server.Receive(data);
    pg = new Paquete(data);
    int respuesta = (int)pg.ObtenerElemento(0);
    if (respuesta == 0)
    {
        server.Shutdown(SocketShutdown.Both);
        server.Close();
        return false;
    }
    else if (respuesta == 1)
    {
        server.Shutdown(SocketShutdown.Both);
        server.Close();
    }

    return true;
}
```

En este código se realiza una conexión con el Servidor de Conexión, se le envía un paquete con el comando de conexión con valor 2 del Servidor de Login, y se espera a la respuesta del Servidor de Conexión que en caso de ser un 0 es que se produjo algún fallo y no se pudo registrar el servidor y en caso de recibir un 1 es que se dio de alta de forma satisfactoria. Finalmente se cierra la conexión establecida con el Servidor de Conexión.

Una vez se ha registrado en el Servidor de Conexión, se ejecuta el método AtenderComandos(), el cual pone al Servidor de Login a la espera de conexiones entrantes.

5.4.2.Proceso de comunicación con jugadores

El proceso de conexión de jugadores será procesado durante la ejecución del método AtenderComandos() que se introdujo en el apartado anterior. Este método tendrá la siguiente codificación:

```
public void AtenderComandos()
{
    int recv;
    byte[] data;
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
puertoTcpRecepcion);

    Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    newsock.Bind(ipep);
    newsock.Listen(10);

    while (true)
    {
        Socket client = newsock.Accept();
        IPEndPoint clientep = (IPEndPoint)client.RemoteEndPoint;

        data = new byte[BYTES];
        recv = client.Receive(data);

        Paquete pg = new Paquete(data);

        int comando = (int)pg.ObtenerElemento(0);
        switch (comando)
        {
            case 0:
                {
//El SC informa de que se va a conectar un nuevo cliente.
//Se devuelve el puerto por el que se le va a atender
                    int identificador = this.CalcularIdentificador() +
1; //+1 porque el identificador empieza en 0
                    int puerto = puertoTcpRecepcion + identificador;
                    Paquete pg2 = new Paquete();
//Comando de asignación de puertos
                    pg2.AgregarElemento(0);
//Puerto asignado
                    pg2.AgregarElemento(puerto);
                    client.Send(pg2.Datos);
//Se crea un nuevo hilo en el que se espera por ese puerto al usuario
                    Thread nuevo = new Thread(new
ThreadStart(this.RecibirCliente));
                    nuevo.Start();
                    break;
                }
            case 1:
                {
//El SC informa de la conexión de un nuevo SPJ y manda su IP+puerto
                    if (_listaSPJ == null)
                        _listaSPJ = new List<ServidorSPJ>();
//Se coge la IP y Puerto
```

```
                    _listaSPJ.Add(new
ServidorSPJ((string)pg.ObtenerElemento(1),
(int)pg.ObtenerElemento(2)));
//Se contesta que todo OK
                    Paquete OK = new Paquete();
                    OK.AgregarElemento(1);
                    client.Send(OK.Datos);
                    break;
                }
        }
        client.Send(data, data.Length, SocketFlags.None);
        client.Close();
    }
    newsock.Close();
}
```

Como se puede ver en el código, el Servidor de Login establece un socket por el que atenderá a las peticiones y según el comando recibido en un paquete de datos entrará por una opción u otra de la instrucción switch. Las dos posibilidades que tiene en el switch son:

- **El Servidor de Conexión envía un 1 para indicar que se ha conectado un Servidor de Juego**

En este caso de forma análoga a la aplicación del Servidor de Conexión, se contará con una clase ServidorSPJ la cual guardará la dirección IP y puerto del Servidor de Login que está estableciendo la conexión, por tanto se cogerán estos datos del paquete recibido desde el Servidor de Conexión para que el Servidor de Login los guarde en su lista de Servidores de Juego _listaSPJ.

- **El Servidor de Conexión envía un 0 para indicar que se quiere conectar un cliente**

Cuando se trata de la conexión de un jugador, es necesario calcular el identificador que se le asignará y para eso se hará una llamada al método CalcularIdentificador() el cual realizará:

```
private int CalcularIdentificador()
{
    int n = 0;
    while (n < ids.Length)
    {
        if (ids[n] == (byte)0)
            break;
        n++;
    }
    return n;
}
```

Es decir, recorrerá el array de bytes en busca de uno con valor 0 (un identificador libre) y devolverá ese identificador más uno para asociárselo al jugador.

A través de este identificador se calculará también el puerto por el que el Servidor de Login ofrecerá servicio al jugador, ya que el puerto se calculará sumando el puerto base de servicio (por el que recibe las peticiones del Servidor de Conexión) más el identificador. Este es el motivo por el que el identificador es el valor de la llamada al método CalcularIdentificador()

más uno, porque sino el puerto asociado podría coincidir con el puerto de servicio ofrecido al Servidor de Conexión.

Una vez definido el puerto por el que se atenderá al cliente, se le devuelve al Servidor de Conexión para que posteriormente este se lo comunique al cliente, y mientras tanto en el Servidor de Login se crea un nuevo hilo de ejecución para atender al cliente que se conectará sin dejar de prestar servicio a las peticiones del Servidor de Conexión.

El nuevo hilo de ejecución comenzará su ejecución en el método RecibirCliente() que tiene la siguiente codificación:

```
private void RecibirCliente()
{
    byte[] datos = new byte[BYTES];
    int identificador = CalcularIdentificador();
    int puerto = puertoTcpRecepcion + identificador + 1;
    ids[identificador] = 1; //Marcamos el identificador como que está
en uso
    try
    {
        IPEndPoint ipepE = new IPEndPoint(IPAddress.Any, puerto);
        Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
        newsock.Bind(ipepE);
        newsock.Listen(10);
        Socket socketEntrada = newsock.Accept();

        //Cerramos el socket que se usa para hacer el bind y
configuramos el nuevo socket
        newsock.Close();
        socketEntrada.NoDelay = true;
        IPEndPoint cliente = (IPEndPoint)socketEntrada.RemoteEndPoint;

        //Obtenemos la IP del SPJ que le atenderá, para saber a quien
informar
        socketEntrada.Receive(datos);
        Paquete pg = new Paquete(datos);
        _spjCliente = (string)pg.ObtenerElemento(0);

        //Le enviamos su identificador
        pg = new Paquete();
        pg.AgregarElemento(identificador);
        socketEntrada.Send(pg.Datos);

        //Recibimos el usuario y contraseña
        datos = new byte[BYTES];
        socketEntrada.Receive(datos);
        Paquete usuario = new Paquete(datos);

        //Comprobamos que el usuario y la contraseña recibidos sean
correctos
        while
(!this.EsUsuarioCorrecto((string)usuario.ObtenerElemento(0),
(string)usuario.ObtenerElemento(1)))
        {
            usuario = new Paquete();
            usuario.AgregarElemento(0); //Login incorrecto
            socketEntrada.Send(usuario.Datos);
```

```
            datos = new byte[BYTES];
            socketEntrada.Receive(datos);
            usuario = new Paquete(datos);
        }

        Jugador j = new Jugador(socketEntrada, identificador,
(string)usuario.ObtenerElemento(0));
        entrada.Add(j);

        //Le enviamos sus personajes
        pg = this.ObtenerPersonajesDelJugador(j);
        socketEntrada.Send(pg.Datos);

        //Recibimos los comandos del jugador
        this.RecibirComandosDeCliente(identificador);
    }
    catch
    {
        this.DesconectarCliente(identificador);
    }
}
```

En este código se puede ver como se espera a la conexión del cliente por el puerto calculado anteriormente y enviado al Servidor de Conexión, y una vez el cliente establece la conexión se sigue el siguiente proceso:

- Se espera a que el cliente envíe el Servidor de Juego que le ha asignado el Servidor de Conexión, para que una vez el jugador seleccione el personaje, el Servidor de Login envíe las coordenadas y resto de datos al Servidor de Juego.
- Se envía el identificador de la sesión al cliente, para que sepa con qué valor está registrado en el Servidor de Login
- Se queda a la espera de que el usuario introduzca en su juego el nombre de usuario y contraseña para validarlo en el sistema
- Una vez se recibe el paquete con el par de datos de usuario y contraseña se verifica si este es válido llamando al método EsUsuarioCorrecto(…), el cual contrastará la información con el fichero de la base de datos de jugadores.
- Si se tiene un par de datos válido, se le envía sus personajes recuperándolos del fichero de la base de datos de jugadores a través del método ObtenerPersonajesDelJugador(…).
- Una vez se han enviado los datos de los personajes del jugador, se comienza la ejecución del método RecibirComandosDeCliente(…) el cual actuará en función de los comandos recibidos, los cuales pueden ser: selección de un personaje, borrado de un personaje y creación de un personaje nuevo.

La codificación de los métodos comentados en el proceso anterior es la siguiente:

```
private bool EsUsuarioCorrecto(string usuario, string pwd)
{
    InterfazESFichero es = new InterfazESFichero(FICHEROPWD);
    return es.Login(usuario, pwd);
}
```

Que llama al método Login de la clase InterfazESFichero que se explicará con mayor detalle en el apartado 5.4.3. Acceso a ficheros.

Respecto al método ObtenerPersonajesDelJugador(…) se tiene la siguiente codificación:

```
private Paquete ObtenerPersonajesDelJugador(Jugador j)
{
    InterfazESFichero es = new InterfazESFichero(FICHEROUSUARIOS);
    if (es.Abrir())
    {
        List<Personaje> lista = es.ObtenerPjs(j.Usuario);
        j.Lista = lista;
        Paquete res = new Paquete();

        res.AgregarElemento(1); //Login correcto

        for (int i = 0; i < lista.Count; i++)
        {
            res.AgregarElemento(lista[i].Nombre);
            res.AgregarElemento(lista[i].Textura);
        }

        es.Cerrar();

        return res;
    }
    else
    {
        return null;
    }
}
```

El método accede al fichero con los personajes del jugador y recupera los nombres de estos y las texturas, para agregarlas al paquete que enviará al jugador para que sepa de qué personajes dispone.

En cuanto al método RecibirComandosDeCliente(…), su codificación será la siguiente:

```
private void RecibirComandosDeCliente(int ide)
{
    byte[] datos = new byte[BYTES];
    int n = -1;
    salir = false;
    while (!salir)
    {
        Jugador j;
        lock (mutex)
        {
            //Localizamos al cliente en el ciclo de ejecución
            for (int i = 0; i < entrada.Count; i++)
            {
                if (entrada[i].Id == ide)
                {
                    n = i;
                    i = entrada.Count;
                }
```

```
            }
            j = entrada[n];
        }
        try
        {
            //Recibimos el comando
            j.Extremo.Receive(datos);
        }
        catch
        {
            this.DesconectarCliente(ide);
            return;
        }
        Paquete pg = new Paquete(datos);

        //Ejecutamos el comando
        EjecutarComando(j, pg);
    }
}
```

En primer lugar se localizará al jugador con el identificador asociado en la lista de jugadores bloqueando el acceso a esta lista para evitar un acceso múltiple al recurso compartido, y una vez se tiene localizado se espera a que envíe algún comando al Servidor de Login.

Recibido el comando se pasará a ejecutar el método EjecutarComando(…) el cual realizará:

```
private void EjecutarComando(Jugador j, Paquete pg)
{
    int comando = (int)pg.ObtenerElemento(1);
    switch (comando)
    {
        case 0:
            {
                //El jugador crea un personaje
                break;
            }
        case 1:
            {
                //El jugador selecciona un personaje
                break;
            }
        case 2:
            {
                //El jugador borra un personaje
                break;
            }
    }
}
```

Como se explicó anteriormente, del jugador se pueden recibir tres tipos de peticiones en el Servidor de Login, la creación, la selección y el borrado de un personaje.

El proceso a seguir en cada caso es el siguiente:

- **Creación de un personaje**

```
case 0:
    {//El jugador crea un personaje
        this.CrearPersonaje(j.Usuario, (string)pg.ObtenerElemento(2));
        Paquete aux = new Paquete();
        aux.AgregarElemento(1); //todo ok
        j.Extremo.Send(aux.Datos);
        break;
    }
```

En el caso de la creación de un personaje se hará una llamada al método CrearPersonaje(…) que realizará:

```
private void CrearPersonaje(string usuario, string pj)
{
    InterfazESFichero es = new InterfazESFichero(FICHEROUSUARIOS);
    Personaje p = new Personaje(pj, "mkcharset", 0, 0);
    es.CrearPj(usuario, p);
}
```

Quedando detallado el método CrearPj(…) en el apartado 5.4.3. Acceso a ficheros.

Finalmente se enviará un paquete al cliente con un valor 1 para indicarle que el proceso fue correctamente.

- **Borrado de un personaje**

```
case 2:
    {//El jugador borra un personaje
        this.BorrarPersonaje(j.Usuario,
(string)pg.ObtenerElemento(2));
        Paquete aux = new Paquete();
        aux.AgregarElemento(1); //todo ok
        j.Extremo.Send(aux.Datos);
        break;
    }
```

En el caso del borrado de un personaje se realizará una llamada al método BorrarPersonaje(…) que tendrá la siguiente codificación:

```
private void BorrarPersonaje(string usuario, string pj)
{
    InterfazESFichero es = new InterfazESFichero(FICHEROUSUARIOS);
    Personaje p = new Personaje(pj, "", 0, 0);
    es.BorrarPj(usuario, p);
}
```

Quedando detallado el método BorrarPj(…) en el apartado 5.4.3. Acceso a ficheros.

Finalmente se enviará un paquete al jugador con valor 1 para indicar que todo fue correctamente.

- **Selección de un personaje**

```
case 1:
    {
        //El jugador selecciona un personaje
        //Comunicacion con SPJ
        Paquete aux = new Paquete();
        //Le decimos que un jugador ha seleccionado un personaje
        aux.AgregarElemento(1);
        //Le decimos la IP
aux.AgregarElemento(((IPEndPoint)j.Extremo.RemoteEndPoint).Address.ToS
tring());
        //Le decimos el usuario
        aux.AgregarElemento(j.Usuario);
        //Le decimos el personaje que ha seleccionado
        aux.AgregarElemento((string)pg.ObtenerElemento(2));

        //Le decimos la textura del personaje

aux.AgregarElemento(j.ObtenerTextura((string)pg.ObtenerElemento(2)));
        //Le pasamos la coordenada X del personaje

aux.AgregarElemento(j.ObtenerCoordenadaX((string)pg.ObtenerElemento(2)
));
        //Le pasamos la coordenada X del personaje

aux.AgregarElemento(j.ObtenerCoordenadaY((string)pg.ObtenerElemento(2)
));

        aux = Soporte.EnviarPaquete(_spjCliente,
this.ObtenerPuertoSPJ(), aux);

        //Comunicación con cliente
        aux = new Paquete();
        aux.AgregarElemento(1); //1 = Todo ok
        j.Extremo.Send(aux.Datos);
        //Si ha seleccionado un personaje, le borramos de la lista de
clientes del SL
        this.DesconectarCliente(j.Id);
        break;
    }
```

El caso de la selección de un personaje difiere de los casos de creación y borrado. Lo primero que se realizará tras recibir el paquete de selección de personaje será enviar un paquete al Servidor de Juego para indicar la información de la selección del jugador, enviando un comando 1 para indicar la selección del jugador, así como la dirección IP del jugador, el nombre de usuario, el nombre del personaje seleccionado, la textura de este personaje y las coordenadas X e Y de su ubicación.

Enviado este paquete al Servidor de Juego, se envía un paquete de vuelta al jugador con un comando 1 para indicar que todo está listo para que se conecte al Servidor de Juego.

Finalmente se procede a la desconexión del cliente en el Servidor de Login con el método DesconectarCliente(…):

```
private void DesconectarCliente(int id)
{
    int n = -1;
      //1. Localizamos al cliente con dicho ID
      for (int i = 0; i < entrada.Count; i++)
      {
            if (entrada[i].Id == id)
            {
                  n = i;
                  i = entrada.Count;
            }
      }
      if (n == -1)
      {
        try
        {
            //Volvemos a activar su identificador
            ids[id] = (byte)0;
            //Cerramos el socket de entrada del cliente y lo quitamos
de la lista de entrada
            entrada[n].Extremo.Shutdown(SocketShutdown.Both);
            entrada[n].Extremo.Close();
            entrada.RemoveAt(n);
        }
        catch
        {
            //No se llegó a establecer la conexión
        }

            return;
      }
      //Cerramos el socket de entrada del cliente y lo quitamos de la
lista de entrada
      entrada[n].Extremo.Shutdown(SocketShutdown.Both);
      entrada[n].Extremo.Close();
      entrada.RemoveAt(n);
      //Volvemos a activar su identificador
      ids[id] = (byte)0;
      //Ponemos la condición de salida a true para que no siga
esperando comandos del cliente
      salir = true;
}
```

De forma resumida, el procedimiento que sigue este método consiste en localizar el identificador del jugador en la lista de jugadores, una vez que se tiene se pone ese identificador a 0 en el array de identificadores para que un nuevo jugador pueda ocupar su posición, y por último se cierran las comunicaciones establecidas con el jugador.

5.4.3.Acceso a ficheros

En el apartado anterior se vieron las llamadas realizadas por la clase ServidorLogin a la clase InterfazESFichero encargada de manejar el acceso a los ficheros, a continuación se detalla el comportamiento de estas clases.

Se tiene una interfaz InterfazES que define los siguientes métodos:

```
public interface InterfazES
{

    bool Abrir();

    bool Cerrar();

    bool Login(string usuario, string passwd);

    List<Personaje> ObtenerPjs(string usuario);

    bool CrearPj(string usuario, Personaje pj);

    bool BorrarPj(string usuario, Personaje pj);

    bool ActualizarPj(Personaje pj);

}
```

La inclusión de esta interfaz se hace con vista a un futuro uso de bases de datos en lugar de ficheros, para que la clase encargada de manejar el acceso a la base de datos tenga el mismo comportamiento que la que actualmente hace el acceso a ficheros.

Para el acceso y manejo del acceso a ficheros se tendrá la clase InterfazESFichero la cual definirá los siguientes atributos y constructor:

```
private string _fichero = null;
private StreamReader _lector = null;

public InterfazESFichero(string nombre)
{
    _fichero = nombre;
}
```

Sus métodos de apertura y cierre de fichero serán:

```
public bool Abrir()
{
    try
    {
        _lector = File.OpenText(_fichero);
        return true;
    }
    catch
    {
        return false;
    }
}
```

```
public bool Cerrar()
{
    try
    {
        _lector.Close();
        return true;
    }
    catch
    {
        return false;
    }
}
```

Por otro lado, para que el acceso a los ficheros sea lo más similar posible, en cuanto a los ficheros de configuración o de almacenamiento de contraseñas se tendrá el siguiente formato:

<clave>: <valor>
<clave>: <valor>
…

En cuanto al formato del fichero de la base de datos de jugadores será como sigue:

<nombre de jugador>: <información de personaje>#<información de personaje>#...
<nombre de jugador>: <información de personaje>#<información de personaje>#...
…

Donde el campo <información de personaje> contendrá:

<nombre de personaje>-<textura>-<coordX>-<coordY>

Las funcionalidades ofrecidas por la clase InterfazESFichero son las siguientes:

- **Recuperar el valor de una clave en un fichero con estructura clave:valor**

```
public string RecuperarCampo(string clave)
{
    String linea = "";
    String res = "";
    while ((linea = _lector.ReadLine()) != null)
    {
        char[] delim = new char[1];
        delim[0] = ':';
        String[] tokens = linea.Split(delim,
StringSplitOptions.RemoveEmptyEntries);
        if (tokens.Length == 0)
            break;
        else if (tokens[0].ToUpper().Equals(clave.ToUpper()))
        {
            res = tokens[1].Trim();
            break;
        }
    }
    return res;
}
```

Recorre las líneas hasta que encuentra a la izquierda de los dos puntos la clave pedida y devuelve el valor de esta clave.

- **Validar a un usuario y contraseña en el sistema**

```
public bool Login(string usuario, string passwd)
{
    bool res = false;
    this.Abrir();

    if (this.RecuperarCampo(usuario).Equals(passwd))
        res = true;

    this.Cerrar();
    return res;
}
```

Devuelve true en caso de que el usuario tenga asociada la contraseña recibida por parámetro.

- **Obtener los personajes de un jugador**

```
public List<Personaje> ObtenerPjs(string usuario)
{
    try
    {
        List<Personaje> res = new List<Personaje>();

        //En el fichero tenemos:
        // usuario: pj1-textura-X-Y#pj2-textura-X-Y#...#
        string todo = this.RecuperarCampo(usuario);

        char[] delim = new char[1];
        delim[0] = '#';
        char[] delim2 = new char[1];
        delim2[0] = '-';

        string[] tokens = todo.Split(delim,
StringSplitOptions.RemoveEmptyEntries);
        string[] pjs;
        for (int i = 0; i < tokens.Length; i++)
        {
            pjs = tokens[i].Split(delim2,
StringSplitOptions.RemoveEmptyEntries);
            Personaje p = new Personaje(pjs[0], pjs[1],
Int32.Parse(pjs[2]), Int32.Parse(pjs[3]));
            res.Add(p);
        }

        return res;
    }
    catch
    {
        return null;
    }
}
```

Recorre el fichero de personajes y guarda la información de estos en objetos de tipo Personaje que posteriormente serán devueltos.

Estos objetos de tipo Personaje tendrán los siguientes atributos y constructor:

```
private string _nombre;
private int _x;
private int _y;
private string _textura;

public Personaje(string nombre, string textura, int x, int y)
{
    _nombre = nombre;
    _textura = textura;
    _x = x;
    _y = y;
}
```

Adicionalmente a esto sólo tendrán propiedades que permitan el acceso a sus campos:

```
public string Textura
{
    get
    {
        return _textura;
    }
}

public int X
{
    get
    {
        return _x;
    }
}

public int Y
{
    get
    {
        return _y;
    }
}

public string Nombre
{
    get
    {
        return _nombre;
    }
}
```

- **Creación de un personaje**

```
public bool CrearPj(string usuario, Personaje pj)
{
    string linea;
    string res = "";
    this.Abrir();

    while ((linea = _lector.ReadLine()) != null)
    {
        if (linea.ToUpper().Contains(usuario.ToUpper()))
        {
            res += linea + pj.Nombre + "-" + pj.Textura + "-625-
675#\n";
        }
        else
        {
            res += linea + "\n";
        }
    }

    this.Cerrar();

    //Volcamos RES al fichero
    StreamWriter salida = new StreamWriter(_fichero, false);
    salida.Write(res);
    salida.Close();

    return true;
}
```

Recuperará del fichero de personajes la línea con los personajes del jugador y añadirá al final al nuevo personaje en las coordenadas definidas como origen de los nuevos jugadores.

- **Borrado de un personaje**

```
public bool BorrarPj(string usuario, Personaje pj)
{
    string linea;
    string res = "";
    this.Abrir();

    while ((linea = _lector.ReadLine()) != null)
    {
        if (linea.ToUpper().Contains(usuario.ToUpper()))
        {
            res += usuario + ": ";
            char[] delim = new char[1];
            delim[0] = ':';
            String[] tokens = linea.Split(delim,
StringSplitOptions.RemoveEmptyEntries);
            //En tokens[1] tenemos a los personajes
            char[] delim2 = new char[1];
            delim2[0] = '#';

            string[] tokens2 = tokens[1].Split(delim2,
StringSplitOptions.RemoveEmptyEntries);
            for (int i = 0; i < tokens2.Length; i++)
            {
```

```
                if
(!tokens2[i].Trim().ToUpper().Contains(pj.Nombre.ToUpper()))
                {//Este le agregamos
                    res += tokens2[i].Trim() + "#";
                }
            }
            res += "\n";
        }
        else
        {
            res += linea.Trim() + "\n";
        }
    }

    this.Cerrar();

    //Volcamos RES al fichero
    StreamWriter salida = new StreamWriter(_fichero, false);
    salida.Write(res);
    salida.Close();

    return true;
}
```

Se accederá al fichero con los personajes del jugador, se recuperará la línea con los personajes del jugador y se irá pasando por ellos y volviendo a agregarlos a excepción del que se recibió para ser borrado.

5.5.Aplicación Servidor de Juego

La aplicación Servidor de Juego se ejecutará en el lado del servidor, y al igual que el Servidor de Login, no será condición indispensable que se ejecute en la misma máquina que ninguno de los servidores anteriormente descritos. Será desarrollada en lenguaje C# como una aplicación de consola ya que no se considera necesario que ningún usuario interactúe con ella a través de una interfaz gráfica.

Será la encargada de dirigir toda la información de la partida, por lo que recibirá la información de cada jugador y una vez validada la enviará al resto de jugadores con el fin de actualizar el estado de la partida en los clientes.

5.5.1.Proceso de alta en el sistema de servidores

El proceso de alta del Servidor de Juego consiste en realizar una petición de registro al Servidor de Conexión el cual le indicará si todo fue correcto según se expone en el apartado 4.2.4. Protocolo de transporte, flujo de datos entre servidores y clientes.

Para ello se tendrá una clase ServidorSPJ con los siguientes atributos y constructor:

```
//Dirección del host
private String host;
//Puertos de recepción de comandos
private int puertoTcpRecepcion;
//Puerto de envío de comandos al Servidor de Conexión
private int puertoTcpEnvioSC;
//Puerto de envío de comandos al cliente
private int puertoTcpEnvioC;
//Nombre del fichero de configuración
static String FICHERO = "./configuracion.txt";
//Listas de conexiones activas
static List<Cliente> entrada;
static List<Cliente> salida;
//Identificadores
static byte[] ids;
//Control de hilos
static bool[] salir;
//TAMAÑO DE ARRAY DE BYTES
private const int BYTES = 1024;
//NUMERO MAXIMO DE JUGADORES
private const int JUGADORES = 2048;
//Instancia de interfaz para el acceso a los jugadores
private InterfazServidorSPJ interfaz;
//Movimientos que pueden realizar los jugadores
public const int CONEXION = 0;
public const int QUIETO = 1;
public const int ARRIBA = 2;
public const int ABAJO = 3;
public const int IZQUIERDA = 4;
public const int DERECHA = 5;
public const int MENSAJECHAT = 6;
public const int SYNC = 10;
public const int DESCONEXION = 11;
public const int ARRIBAIZQUIERDA = 12;
```

```
public const int ARRIBADERECHA = 13;
public const int ABAJOIZQUIERDA = 14;
public const int ABAJODERECHA = 15;
//Lista de personajes que se están conectando (para controlar los pjs
que han seleccionado)
static List<ClienteConectando> _listaConectando = null;
//Variable para mecanismo de sincronización en el acceso a variables
compartidas
static object mutex;
//Linea de comandos
private bool mostrar;
#endregion

public ServidorSPJ(InterfazServidorSPJ inter)
{
    entrada = new List<Cliente>();
    salida = new List<Cliente>();
    ids = new byte[JUGADORES];
    salir = new bool[JUGADORES];
    for (int i = 0; i < JUGADORES; i++)
    {
        salir[i] = false;
    }

    host = ObtenerDatoDeFichero("host");
    puertoTcpRecepcion = Int32.Parse(ObtenerDatoDeFichero("tcp_r"));
    puertoTcpEnvioSC = Int32.Parse(ObtenerDatoDeFichero("tcp_s_sc"));
    puertoTcpEnvioC = Int32.Parse(ObtenerDatoDeFichero("tcp_s_c"));

    mutex = new object();
    mostrar = true;
    interfaz = inter;
}
```

En cuanto al método Main(…) que iniciará la ejecución de la aplicación se tiene la siguiente codificación:

```
static void Main(string[] args)
{
    InterfazServidorSPJ interfaz = new InterfazServidorSPJ();
    interfaz.ArrancarServidor();
    string comando;
    while (true)
    {
        comando = Console.ReadLine();
        interfaz.EjecutarComando(comando);
    }
}
```

Este método Main(…) difiere de las aplicaciones de los otros servidores en que tendrá además del hilo de ejecución que recibe las peticiones de servicio, y de los hilos por los que está presentando servicio a los jugadores, un hilo adicional a través del cual se pueden introducir comandos al servidor para obtener datos como el número de usuarios conectados actualmente, datos de éstos, etc.

En cuanto a la InterfazServidorSPJ, será la clase encargada de intermediar entre los comandos introducidos en la consola y el ServidorSPJ que dirigirá la lógica del servidor.

Sus atributos y constructor tendrán la siguiente codificación:

```
//Instancia del servidor de juego
private ServidorSPJ spj;
//Lista de jugadores
private List<Jugador> jugadores;

public InterfazServidorSPJ()
{
    jugadores = new List<Jugador>();
}
```

De modo que cuando en el método Main(…) se realice la llamada al método ArrancarServidor() de la clase InterfazServidorSPJ, se ejecutará lo siguiente:

```
public void ArrancarServidor()
{
    spj = new ServidorSPJ(this);

    if (spj.RegistrarEnSC())
    {
        Thread nuevo = new Thread(new
ThreadStart(spj.AtenderComandos));
        nuevo.Start();
    }
    else
    {
        Console.ReadLine();
    }
}
```

Como se puede ver, es en la llamada a este método donde se crea el primer hilo de ejecución paralelo a la ejecución del hilo inicial, quedando la ejecución del hilo inicial para la ejecución de comandos desde consola, y el nuevo hilo para atender a los comandos recibidos en el servidor desde otros servidores o clientes.

También es en el método ArrancarServidor() donde se realiza el registro del servidor al llamar al método RegistrarEnSC() de la clase ServidorSPJ que tiene la siguiente codificación:

```
public bool RegistrarEnSC()
{
    byte[] data;
    IPHostEntry address = Dns.GetHostEntry(host);
    IPEndPoint ipep = new IPEndPoint(Soporte.DevolverIPv4(address),
puertoTcpEnvioSC);
    Socket server = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    server.NoDelay = true;

    try
    {
        server.Connect(ipep);
    }
    catch (SocketException e)
    {
        return false;
    }
```

```
        //Conexión
        //Comando de conexion de SPJ en SC-->3
        Paquete pg = new Paquete();
        pg.AgregarElemento(3); //Comando 3, conexión de SPJ en SC
        pg.AgregarElemento(puertoTcpRecepcion);
        server.Send(pg.Datos);

        data = new byte[BYTES];

        int recv = server.Receive(data);
        pg = new Paquete(data);
        int respuesta = (int)pg.ObtenerElemento(0);
        if (respuesta == 0)
        {
            return false;
        }
        else if (respuesta == 1)
        {
            server.Shutdown(SocketShutdown.Both);
            server.Close();
        }

        return true;
    }
```

Como se puede ver, una vez conectado con el Servidor de Conexión del cual se conoce su IP y puerto de servicio recuperándolo a través del fichero de configuración, se le envía un paquete con el comando 3 además del puerto por el que el Servidor de Juego ofrecerá servicio. Si la respuesta del Servidor de Conexión a esta petición de registro es 0 entonces ha fallado el proceso de alta del Servidor de Juego y no se comenzará a prestar servicio, y en caso de recibir un 1 se cerrará la conexión y se dará paso a la ejecución del método AtenderComandos() que permitirá la recepción de nuevos jugadores a la partida.

5.5.2.Proceso de conexión de jugadores

El proceso de conexión de un nuevo jugador al Servidor de Juego se divide en tres partes:

- **El Servidor de Conexión informa de que un jugador se ha conectado a él**, en este punto el Servidor de Juego tiene que preparar un nuevo hilo de ejecución y un puerto por donde atenderá a ese jugador cuando se conecte a él.
- **El Servidor de Login informa de que el jugador ha seleccionado un determinado personaje**, así que envía al Servidor de Juego los datos del personaje para que cuando el jugador se conecte ya se tenga la información (nombre del usuario, nombre del personaje, textura y coordenadas)
- **El jugador se conecta al Servidor de Juego** y comienza la partida

El código encargado de ofrecer esta funcionalidad se encuentra dentro del método AtenderComandos() de la clase ServidorSPJ que será ejecutado tras realizar el proceso de alta en el Servidor de Conexión como se explicó en el apartado anterior.

El código de este método es el siguiente:

```
public void AtenderComandos()
{
    int recv;
    byte[] data;
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
puertoTcpRecepcion);

    Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

    newsock.Bind(ipep);
    newsock.Listen(10);

    while (true)
    {
        Socket client = newsock.Accept();
        IPEndPoint clientep = (IPEndPoint)client.RemoteEndPoint;
        data = new byte[BYTES];
        recv = client.Receive(data);
        Paquete pg = new Paquete(data);
        int comando = (int)pg.ObtenerElemento(0);

        switch (comando)
        {
            case 0:
                {//El SC nos informa de que se va a conectar un nuevo
cliente.
                    //Le devolvemos el puerto por el que le vamos a
atender
                    int identificador = this.CalcularIdentificador() +
1; //+1 porque el identificador empieza en 0
                    int puerto = puertoTcpRecepcion + identificador;
                    Paquete pg2 = new Paquete();
                    pg2.AgregarElemento(0); //0 es el comando para
asignar puertos
                    pg2.AgregarElemento(puerto);
```

```
                    client.Send(pg2.Datos);

                    //Creamos un nuevo hilo en el que esperamos por
ese puerto al usuario
                    Thread nuevo = new Thread(new
ThreadStart(this.RecibirCliente));
                    nuevo.Start();
                    break;
                }
            case 1:
                {//El SL nos informa de que se va a conectar un
usuario por una IP con un personaje
                    if (_listaConectando == null)
                        _listaConectando = new
List<ClienteConectando>();
                    string ipCliente = (string)pg.ObtenerElemento(1);
                    string nombreUsuario =
(string)pg.ObtenerElemento(2);
                    string pjSeleccionado =
(string)pg.ObtenerElemento(3);
                    string textura = (string)pg.ObtenerElemento(4);
                    int coordX = (int)pg.ObtenerElemento(5);
                    int coordY = (int)pg.ObtenerElemento(6);

                    _listaConectando.Add(new
ClienteConectando(ipCliente, nombreUsuario, pjSeleccionado, textura,
coordX, coordY));
                    //Le decimos que todo OK a SL (1)
                    Paquete OK = new Paquete();
                    OK.AgregarElemento(1);
                    client.Send(OK.Datos);
                    break;
                }
        }
        client.Send(data, data.Length, SocketFlags.None);
        client.Close();
    }
    newsock.Close();
}
```

Como se puede ver en el código, el método AtenderComandos() abre un socket que deja a la escucha por el puerto que se le indicó en el fichero de configuración para después entrar en un bucle de servicio a través del cual recibirá los comandos con las peticiones que recibirá del Servidor de Conexión (indicará que un nuevo jugador se va a conectar) y del Servidor de Login (indicará que personaje ha elegido este jugador).

Estas dos peticiones se identifican en el switch, si se trata de una petición del Servidor de Conexión, el Servidor de Juego recibirá un paquete con el comando 0, y en el caso de una petición del Servidor de Login se recibirá un comando 1.

Según se comentó en el primero punto al inicio del apartado, cuando un jugador inicie una partida lo primero que se producirá en el Servidor de Juego será que reciba un paquete con comando 0 desde el Servidor de Conexión, entonces el Servidor de Juego calculará un identificador para asignarle al nuevo jugador que inicia partida, y este cálculo lo hará igual que lo hacía el Servidor de Login tal y como se explicó en el apartado 5.4.2. Proceso de comunicación con jugadores.

A través de este puerto el Servidor de Juego calculará el puerto por el que ofrecerá servicio al jugador (el puerto será la suma del identificador más el puerto base por donde atiende a los servidores), se lo enviará en un Paquete al Servidor de Conexión para que se lo comunique al jugador e iniciará un nuevo hilo de ejecución sobre el método RecibirCliente() que será el que maneje la comunicación con ese jugador.

En cuanto al segundo punto del inicio del apartado, cuando el jugador haya seleccionado un personaje con el que jugar en el Servidor de Login, este servidor le enviará un Paquete con comando 1 al Servidor de Juego para enviarle los datos del personaje seleccionado.

En este Paquete los datos que se reciben son: la IP del jugador, el nombre del usuario, el nombre del personaje, la textura y las coordenadas X e Y del personaje.
Con estos datos, el Servidor de Juego agrega a la lista de los jugadores que se están conectando al nuevo jugador, y le envía un paquete al Servidor de Login con comando 1 para indicar que el proceso se ha realizado satisfactoriamente.

Llegado a este punto, el Servidor de Juego ya tiene todos los datos que necesita del jugador y tiene un hilo de ejecución preparado para atenderle por lo que sólo queda que el jugador realice la conexión a este tras cerrar la conexión con el Servidor de Login. Este es el tercer punto que se exponía al principio del apartado.

Para manejar la conexión realizada por el usuario se tendrá el método RecibirCliente() que se estaba ejecutando en el hilo de ejecución paralelo y que tiene la siguiente codificación:

```
private void RecibirCliente()
{
    int identificador = CalcularIdentificador();
    ids[identificador] = 1; //Marcamos el identificador como que está
en uso
    int puerto = puertoTcpRecepcion + identificador + 1;
    IPEndPoint ipepE = new IPEndPoint(IPAddress.Any, puerto);
    Socket newsock = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    newsock.Bind(ipepE);
    newsock.Listen(10);

    Socket socketEntrada = newsock.Accept();
    lock (mutex)
    {
        #region Establecimiento de conexión dirección Cliente-
>Servidor. Se guarda en la lista entrada.
        //Cerramos el socket que se usa para hacer el bind y
configuramos el nuevo socket
        newsock.Close();
        socketEntrada.NoDelay = true;
        IPEndPoint cliente = (IPEndPoint)socketEntrada.RemoteEndPoint;

        byte[] datos = new byte[BYTES];
        //Localizamos el id y el nombre del usuario
        string id_pj = this.ObtenerIdPj(cliente.Address.ToString());
        ClienteConectando cc = null;
        for (int i = 0; i < _listaConectando.Count; i++)
        {
```

```
            if
(_listaConectando[i].PjSeleccionado.ToUpper().Equals(id_pj.ToUpper()))
            {
                cc = _listaConectando[i];
                break;
            }
        }
        string usuario =
this.ObtenerUsuario(cliente.Address.ToString());
        Cliente je = new Cliente(socketEntrada, identificador);
        //Le enviamos su identificador
        Paquete pg = new Paquete();
        pg.AgregarElemento(identificador);
        socketEntrada.Send(pg.Datos);

        entrada.Add(je);

        //Agramos al jugador a la interfaz
        List<int> coords = new List<int>();
        coords.Add(cc.X);
        coords.Add(cc.Y);
        interfaz.AgregarJugador(id_pj, identificador, coords,
cc.Textura);
        #endregion

        #region Establecimiento de conexión dirección Servidor-
>Cliente. Se guarda en la lista salida.
        IPEndPoint ipepS = new IPEndPoint(cliente.Address,
puertoTcpEnvioC);
        Socket socketSalida = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
        socketSalida.NoDelay = true;

        try
        {
            socketSalida.Connect(ipepS);
        }
        catch (SocketException e)
        {
            return;
        }

        Cliente js = new Cliente(socketSalida, identificador);

        salida.Add(js);
        #endregion

        #region Se le informa de los otros jugadores que ya están en
el servidor

        List<Jugador> jugadores = interfaz.ObtenerJugadores();
        Paquete aux;
        for (int i = 0; i < jugadores.Count; i++)
        {
            if (jugadores[i].Id != identificador)
            {
                aux = new Paquete();

                jugadores[i].CalcularPosicion();
                Coordenada pos = jugadores[i].PosicionMovimiento;
                aux.AgregarElemento(1); //Hay un jugador
```

```
                    aux.AgregarElemento(jugadores[i].Nombre);
                    aux.AgregarElemento((int)pos.x);
                    aux.AgregarElemento((int)pos.y);
                    aux.AgregarElemento(jugadores[i].Id);
                    aux.AgregarElemento(jugadores[i].Movimiento);
                    aux.AgregarElemento(jugadores[i].Textura);
                    js.Extremo.Send(aux.Datos);
                }
                else
                {
                    aux = new Paquete();
                    aux.AgregarElemento(1); //Hay un jugador
                    aux.AgregarElemento(jugadores[i].Nombre);
                    aux.AgregarElemento(coords[0]);
                    aux.AgregarElemento(coords[1]);
                    aux.AgregarElemento(identificador);
                    aux.AgregarElemento(QUIETO);
                    aux.AgregarElemento(jugadores[i].Textura);
                    js.Extremo.Send(aux.Datos);
                }
                datos = new byte[BYTES];
                je.Extremo.Receive(datos);
            }

            aux = new Paquete();
            aux.AgregarElemento(0);//No hay más jugadores
            js.Extremo.Send(aux.Datos);

            datos = new byte[BYTES];
            je.Extremo.Receive(datos);

            #endregion
        }

        this.RecibirComandosDeCliente(identificador);
    }
```

El código del método RecibirCliente() se estructura de la siguiente forma:

- Creación de un socket al que se conecta el cliente por el puerto calculado a través de su identificador y posterior conexión del jugador al socket. Por este socket el jugador enviará sus movimientos
- Una vez conectado el jugador, se recupera su información de la lista de jugadores que se están conectando, se le envía su identificador para confirmar la conexión y se agrega al jugador a la lista de jugadores conectados en la partida de la clase InterfazServidorSPJ.
- Creación de un socket desde el que el Servidor de Juego se conectará al jugador. Por este socket el Servidor enviará al Jugador los movimientos del resto de jugadores.
- Una vez conectado al jugador, el Servidor de Juego recorrerá la lista de los jugadores conectados y le enviará los datos de estos (nombres, coordenadas X e Y, texturas y movimientos).

Alcanzado este punto, ya se tiene al jugador conectado al Servidor de Juego con todos los elementos necesarios para la comunicación preparados para la partida, así que en el mismo hilo de ejecución desde el que se ha realizado todo este proceso, se llama al método

RecibirComandosDeCliente(…) para que se encargue de recibir todos los movimientos y acciones que realice el jugador para comunicárselos al resto de jugadores. Este proceso será tratado en el apartado 5.5.3. Broadcasting de información.

5.5.3.Broadcasting de información

Una vez el cliente se ha conectado al Servidor de Juego comenzará la partida. En ese momento empezará a enviar Paquetes con los movimientos que está realizando para que éste los reenvíe al resto de jugadores y viceversa.

De este proceso se encargará el método RecibirComandosDeCliente(…), que se estará ejecutando en un hilo de ejecución paralela al hilo principal del Servidor de Juego, y del que habrá tantos hilos como jugadores haya conectados a la vez. El código realizado por este método es el siguiente:

```
private void RecibirComandosDeCliente(int ide)
{
    byte[] datos = new byte[BYTES];
    int n = -1;
    lock (mutex)
    {
        salir[ide] = false;
    }
    Paquete anterior = null;
    while (!salir[ide])
    {
        //Localizamos al cliente en el ciclo de ejecución
        Cliente j;
        lock (mutex)
        {
            for (int i = 0; i < entrada.Count; i++)
            {
                if (entrada[i].Id == ide)
                {
                    n = i;
                    i = entrada.Count;
                }
            }
            j = entrada[n];
        }

        //Recibimos el comando
        try
        {
            j.Extremo.Receive(datos);

            lock (mutex)
            {
                //Ejecutamos el comando
                Paquete pg = new Paquete(datos);
                if ((int)pg.ObtenerElemento(1) == MENSAJECHAT)
                {
                    int x = 0;
                }
                this.EjecutarComando(pg);
                anterior = pg;
            }
        }
        catch
        {
            lock (mutex)
            {
```

```
                //Si el cliente cierra la conexión de golpe entra
aquí.
                Paquete pgdes = new Paquete();
                pgdes.AgregarElemento(ide);
                pgdes.AgregarElemento(DESCONEXION);
                this.BroadcastPaquete(pgdes);
                return;
            }
        }
    }
}
```

Lo primero que realiza el método es poner el valor false en el array de salida en la posición del identificador del jugador manejado por este hilo de ejecución, este valor false indica que el jugador no se va a desconectar en las siguientes iteraciones y sólo cambiará cuando el jugador realice un proceso de desconexión, que será explicado en el apartado 5.5.5. Proceso de desconexión de jugadores.

A continuación localizará al jugador de entre la lista de clientes bloqueándola como mecanismo de seguridad para evitar que durante su búsqueda cambie el estado debido al acceso compartido a esta variable desde los múltiples hilos de ejecución.

Una vez localizado al jugador en la lista, el hilo quedará a la espera de recibir un nuevo movimiento del jugador. En este punto pueden suceder dos cosas:

- Se cierra el socket debido a que el jugador ha finalizado la partida finalizando el proceso del juego sin escoger la opción de cerrar sesión. Este proceso será explicado en 5.5.5. Proceso de desconexión de jugadores.
- Se recibe un movimiento y se manda al método EjecutarComando(…)

La codificación del método EjecutarComando(…) es la siguiente:

```
private void EjecutarComando(Paquete pg)
{
    pg = interfaz.ActualizarJugador(pg);
    this.BroadcastPaquete(pg);
}
```

Como se puede ver, el código ejecuta el método ActualizarJugador(…) sobre la interfaz que contiene la lista de jugadores con sus datos que actualizará coordenadas y movimiento de éste, y después realizará el “broadcast” del Paquete, que no es un broadcast por definición ya que si es necesario realizar un envío una vez por cada cliente aunque la funcionalidad final que persigue es similar.

En cuanto al método ActualizarJugador(…) se tiene la siguiente codificación:

```
public Paquete ActualizarJugador(Paquete pg)
{
    int id = (int)pg.ObtenerElemento(0);
    int movi = (int)pg.ObtenerElemento(1);
    Paquete res = new Paquete();
```

```
    res.AgregarElemento(id);
    res.AgregarElemento(movi);
    if (movi == 6)
    {
        //En este caso se recibe un mensaje de chat, no hay que
actualizar a ningún jugador
        string s = (string)pg.ObtenerElemento(2);
        res.AgregarElemento(s);
    }
    else
    {
        //Localizamos el jugador
        Jugador j = null;
        for (int i = 0; i < jugadores.Count; i++)
        {
            if (jugadores[i].Id == id)
            {
                j = jugadores[i];
                i = jugadores.Count;
            }
        }
        if (j != null)
        {
            int x = (int)pg.ObtenerElemento(2);
            int y = (int)pg.ObtenerElemento(3);
            j.FinalizarMovimiento(movi, x, y);
            res.AgregarElemento((int)j.PosicionAbsoluta.x);
            res.AgregarElemento((int)j.PosicionAbsoluta.y);
        }
    }
    return res;
}
```

Al recibir el paquete el método ActualizarJugador(…), comprueba que tipo de movimiento ya que si se trata de un mensaje de chat no será necesario actualizar los datos del jugador, en caso de que se trate de otro tipo de acción (un movimiento hacia alguna dirección), se localizará al jugador en la lista de jugadores, se obtendrán las coordenadas ya que el tipo de movimiento ya se tiene, y se finalizará el movimiento actualizando las coordenadas de posición de éste a través del método FinalizarMovimiento(…).

Finalmente se agregarán las nuevas coordenadas calculadas del jugador junto con el identificador y movimiento realizador por éste para su posterior “broadcast”.

La codificación del método FinalizarMovimiento(…) es la siguiente:

```
public void FinalizarMovimiento(int nuevoMovimiento, int x, int y)
{
    posicionAbsoluta.x = x;
    posicionAbsoluta.y = y;
    this.movimiento = nuevoMovimiento;
    this.instante = DateTime.Now;
}
```

El Paquete creado en ActualizarJugador(…) es devuelto al método EjecutarComando(…) para realizar el proceso de broadcast a través del método BroadcastPaquete(…) que tiene la siguiente codificación:

```
private void BroadcastPaquete(Paquete pg)
{
    int id = (int)pg.ObtenerElemento(0);
    int mov = (int)pg.ObtenerElemento(1);
    Jugador jugador = interfaz.ObtenerJugador(id);

    switch (mov)
    {
        case CONEXION:
            {
                pg.AgregarElemento(jugador.Nombre);
                pg.AgregarElemento(jugador.Movimiento);
                pg.AgregarElemento(jugador.Textura);
                break;
            }
        case DESCONEXION:
            {
                this.DesconectarCliente(id);
                break;
            }
        default:
            {
                break;
            }
    }

    //Reenviamos el paquete a todos los jugadores conectados
    for (int i = 0; i < salida.Count; i++)
    {
        salida[i].Extremo.Send(pg.Datos);
    }
}
```

Este código lo primero que realiza es recuperar los datos del jugador del que va a enviar el paquete a través del identificador con el siguiente método:

```
public Jugador ObtenerJugador(int id)
{
    Jugador j = null;
    for (int i = 0; i < jugadores.Count; i++)
    {
        if (jugadores[i].Id == id)
        {
            j = jugadores[i];
            i = jugadores.Count;
        }
    }
    return j;
}
```

A continuación si el paquete es de tipo conexión agregará al paquete el nombre del jugador, movimiento y textura, si es de desconexión ejecutará el método de DesconectarCliente(…), y sea cual sea el tipo de paquete, realizará finalmente el proceso de

“broadcast” que consistirá en recorrer la lista de jugadores enviando el paquete recibido y modificado si se trataba de una conexión.

5.5.4.Seguridad e integridad de la información transmitida

Dado que se pueden producir errores en la transmisión de la información o compartimientos maliciosos por parte de los jugadores, los paquetes deberían ser inspeccionados antes de actualizar la información de los jugadores y realizar el broadcast de la información recibida.

Por ello en el diseño de la aplicación Servidor de Juego se separo en un método aparte estos dos procedimientos, el método fue expuesto en el apartado anterior: EjecutarComando(…). Este método se ejecuta a cada paquete recibido por los jugadores y es el encargado de actualizar al jugador con la información que le ha enviado en el paquete y de realizar el broadcast con la nueva información actualizada del jugador como se veía en su codificación:

```
private void EjecutarComando(Paquete pg)
{
    pg = interfaz.ActualizarJugador(pg);
    this.BroadcastPaquete(pg);
}
```

De modo que en caso de tener que introducirse un mecanismo que garantice la seguridad e integridad de los datos debería ser antes de realizar estas dos instrucciones contenidas en EjecutarComando(…).

Un ejemplo de confirmación de paquetes para garantizar la estructura de los datos recibidos (aunque no la posibilidad de sus valores) sería la siguiente:

```
private void EjecutarComando(Paquete pg)
{
    if (this.ConfirmarPaquete(pg))
    {
        pg = interfaz.ActualizarJugador(pg);
        this.BroadcastPaquete(pg);
    }
}

private bool ConfirmarPaquete(Paquete pg)
{
    bool res = true;
    try
    {
        int id = (int)pg.ObtenerElemento(0);
        int mov = (int)pg.ObtenerElemento(1);
        switch (mov)
        {
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
                int x = (int)pg.ObtenerElemento(2);
                int y = (int)pg.ObtenerElemento(3);
                break;
            case 6:
                string mensaje = (string)pg.ObtenerElemento(2);
```

```
                break;
            default:
                break;
        }
    }
    catch (Exception ex)
    {
        res = false;
    }
    return res;
}
```

Cada paquete que se recibiera pasaría por el método ConfirmarPaquete(…) que sólo devolverá true en caso de que la estructura del Paquete fuera la correcta para el tipo de movimiento, siendo las acciones del 1 al 5 quieto, arriba, abajo, izquierda y derecha; y la acción 6 enviar un mensaje de chat.

5.5.5.Proceso de desconexión de jugadores

Como se explicó en el apartado 5.5.3. Broadcasting de información, hay dos formas de que el jugador se desconecte del sistema:

- Desconexión negociada, el jugador envía un paquete al Servidor de Juego solicitando la desconexión.
- Desconexión sin negociar, el jugador finaliza la aplicación sin enviar el paquete de desconexión al Servidor de Juego y éste tiene que controlar el evento.

Para el primer caso el Servidor de juego recibirá del jugador un paquete con código de acción con valor 11, a lo que el servidor actuará como si de un paquete normal se tratara hasta realizar el broadcast en el método BroadcastPaquete(…) donde comprobará si el paquete que va a enviar al resto de jugadores es una desconexión, y en cuyo caso ejecutará el método DesconectarCliente(…) con la siguiente codificación:

```
private void DesconectarCliente(int id)
{
    int n = -1;
      //1. Localizamos al cliente con dicho ID
      for (int i = 0; i < entrada.Count; i++)
      {
        if (entrada[i].Id == id)
            {
                  n = i;
                  i = entrada.Count;
            }
      }
      if (n == -1)
      {
            return;
      }
     ((IPEndPoint)entrada[n].Extremo.RemoteEndPoint).Address);
      //Cerramos el socket de entrada del cliente y lo quitamos de la
lista de entrada
      entrada[n].Extremo.Shutdown(SocketShutdown.Both);
      entrada[n].Extremo.Close();
      entrada.RemoveAt(n);
      //Cerramos el socket de salida del cliente y lo quitamos de la
lista de salida
      salida[n].Extremo.Shutdown(SocketShutdown.Both);
      salida[n].Extremo.Close();
      salida.RemoveAt(n);
    //Lo eliminamos de la interfaz
    interfaz.EliminarJugador(id);
      //Volvemos a activar su identificador
      ids[id] = (byte)0;
      //Ponemos la condición de salida a true para que no siga
esperando comandos del cliente
      salir[id] = true;
}
```

El método DesconectarCliente(…) recibe el identificador del cliente que se va a desconectar, de modo que lo primero que realiza es localizar al jugador en la lista de jugadores, y una vez tiene el índice de este jugador en la lista, cierra las dos comunicaciones establecidas con el jugador (una en dirección servidor->cliente y otra en dirección cliente->servidor).

Finalmente elimina al jugador de la lista de jugadores de la interfaz encargada de actualizar los datos, pone su identificador a 0 para que pueda ser asignado a un futuro jugador y establece en el array de condición de salida que ese jugador ya va finalizar su ciclo de ejecución, así en la siguiente ejecución del método RecibirComandosDeCliente(…) se detendrá el bucle encargado de quedar a la espera de paquetes del jugador y de este modo finalizará también la ejecución del hilo creado para dicho jugador.

La otra posibilidad de desconexión se realiza sin envío de paquete de desconexión por parte del jugador y se produce cuando este cierra el juego finalizando el proceso al presionar sobre el botón de cerrar de la ventana de la aplicación.
Este evento se capturará en el bloque try…catch del método RecibirComandosDeCliente(…), y en caso de capturarse procederá con la siguiente ejecución:

```
lock (mutex)
{
    //Si el cliente cierra la conexión de golpe entra aquí.
    Paquete pgdes = new Paquete();
    pgdes.AgregarElemento(ide);
    pgdes.AgregarElemento(DESCONEXION);
    this.BroadcastPaquete(pgdes);
    return;
}
```

Es decir, si se detecta que el jugador ha cerrado la aplicación, el servidor creará un paquete de desconexión que enviará un paquete al resto de jugadores y que al entrar por el método BroadcastPaquete(…) implicará la ejecución y explicación anterior del método DesconectarCliente(…) dándole de baja en el sistema de juego.

6.Gestión del proyecto

6.1.Diagrama Gantt

A continuación se incluyen las planificaciones realizadas para el proyecto.

En primer lugar se muestra la planificación que se estimó para el proyecto en forma de diagrama Gantt, las tareas identificadas, el orden en que iban a ser abordadas y su duración estimada.

En segundo lugar se muestra la planificación una vez finalizado el proyecto, en el que realizando un proceso de seguimiento sobre el diagrama Gantt estimado se fueron estableciendo las duraciones reales para las tareas.

En cuanto a las tareas, se dividen en agrupaciones basadas en delimitaciones lógicas para la resolución del proyecto, de modo que se tiene:

- **Análisis del problema**

Desglose de los sub-problemas que un videojuego multijugador masivo tiene que resolver, indicando y estudiando las diferentes opciones que ofrecen las tecnologías actuales.

- **Diseño de solución**

Propone distintos diseños que solucionan el problema para su prueba en la práctica y elección final del más eficiente.

- **Implementación**

Definen las tareas en las que se codificarán las soluciones propuestas en el Diseño de la solución.

Se divide en tres subgrupos:

- **Implementación y prueba de prototipos**, en la que se codifican los diseños propuestos a modo de prototipo para comprobar su eficiencia y seleccionar el sistema final
- **Desarrollo de sistema de servidores final**, en el que ya se ha seleccionado la mejor solución y se codifica todo lo necesario para poner en marcha el sistema de servidores del modelo cliente-servidor que dará solución al problema
- **Desarrollo de aplicación cliente final en XNA**, que será la codificación de la aplicación cliente basándose en la plataforma XNA.

- **Documentación**

Define las tareas cuyo objetivo es la documentación de cada uno de los pasos lógicos realizados durante el proyecto.

En cuanto a la secuenciación, durante el desarrollo de cada una de las tareas de **Análisis del problema**, **Diseño de la solución** e **Implementación** se realizará de forma paralela su correspondiente tarea de documentación, y puesto el que el orden lógico es Análisis, Diseño e Implementación, no se comenzará con el siguiente apartado hasta no haber terminado el anterior.

6.1.1.Planificación Gantt estimada

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Estudio de documentación**	73 días?	mié 13/01/10	vie 23/04/10	
2		**Creación de memoria y documentación**	**73 días?**	mié 13/01/10	vie 23/04/10	
3		Estado de la cuestión	15 días?	mié 13/01/10	mar 02/02/10	
4		Análisis del problema	10 días?	mié 03/02/10	mar 16/02/10	3
5		Solución del problema	20 días?	mié 17/02/10	mar 16/03/10	4
6		Desarrollo de sistema de servidores	10 días?	mié 17/03/10	mar 30/03/10	5
7		Desarrollo de aplicación XNA	15 días?	mié 31/03/10	mar 20/04/10	6
8		Introducción	1 día?	mié 21/04/10	mié 21/04/10	7
9		Apuntes finales	2 días?	jue 22/04/10	vie 23/04/10	8
10		**Análisis del problema**	**7 días?**	mié 03/02/10	**jue 11/02/10**	**4CC**
11		Impacto de diferentes tipos de juego	1 día?	mié 03/02/10	mié 03/02/10	**4CC**
12		Sistemas de almacenamiento	1 día?	jue 04/02/10	jue 04/02/10	11
13		Arquitecturas de red	2 días?	vie 05/02/10	lun 08/02/10	12
14		Protocolos de comunicación de nivel de transporte	2 días?	mar 09/02/10	mié 10/02/10	13
15		Concurrencia	1 día?	jue 11/02/10	jue 11/02/10	14
16		**Diseño de solución**	**10 días?**	mié 17/02/10	**mar 02/03/10**	**5CC**
17		Sistema de servidores y cliente	2 días?	mié 17/02/10	jue 18/02/10	**5CC**
18		Secuenciación en el protocolo de comunicación	1 día?	vie 19/02/10	vie 19/02/10	17
19		Modelo Cliente-Servidor UDP no concurrente	1 día?	lun 22/02/10	lun 22/02/10	18
20		Modelo Cliente-Servidor híbrido TCP/UDP concurrente	1 día?	mié 24/02/10	mié 24/02/10	24
21		Modelo Cliente-Servidor TCP no concurrente	1 día?	vie 26/02/10	vie 26/02/10	25
22		Modelo Cliente-Servidor TCP concurrente	1 día?	mar 02/03/10	mar 02/03/10	26

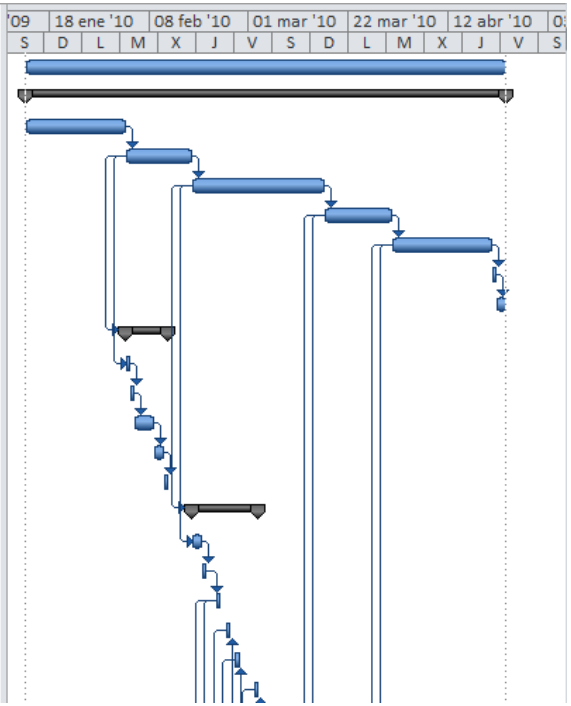

Imagen 49. Diagrama de Gantt - Planificación estimada 1/3

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
23		**Implementación y pruebas de prototipos**	**8 días?**	lun 22/02/10	mié 03/03/10	**19CC**
24		Modelo Cliente-Servidor UDP no concurrente	2 días?	lun 22/02/10	mar 23/02/10	19CC
25		Modelo Cliente-Servidor híbrido TCP/UDP concurrente	2 días?	mié 24/02/10	jue 25/02/10	20CC
26		Modelo Cliente-Servidor TCP no concurrente	2 días?	vie 26/02/10	lun 01/03/10	21CC
27		Modelo Cliente-Servidor TCP concurrente	2 días?	mar 02/03/10	mié 03/03/10	22CC
28		**Desarrollo de sistema de servidores final**	**14 días?**	mié 17/03/10	**lun 05/04/10**	**6CC**
29		**Servidor de conexión**	**4 días?**	mié 17/03/10	**lun 22/03/10**	**6CC**
30		Conexión y desconexión de servidores	2 días?	mié 17/03/10	jue 18/03/10	**6CC**
31		Conexión de clientes	2 días?	vie 19/03/10	lun 22/03/10	30
32		**Servidor de login**	**4 días?**	**mar 23/03/10**	**vie 26/03/10**	**29**
33		Alta en el sistema de servidores	1 día?	mar 23/03/10	mar 23/03/10	29
34		Comunicación con jugadores	2 días?	mié 24/03/10	jue 25/03/10	33
35		Acceso a sistema de almacenamiento	1 día?	vie 26/03/10	vie 26/03/10	34
36		**Servidor de juego**	**6 días?**	**lun 29/03/10**	**lun 05/04/10**	**32**
37		Alta en el sistema de servidores	1 día?	lun 29/03/10	lun 29/03/10	**32**
38		Conexión de jugadores	1 día?	mar 30/03/10	mar 30/03/10	37
39		Broadcasting de información	2 días?	mié 31/03/10	jue 01/04/10	38
40		Seguridad e integridad de la información transmitida	1 día?	vie 02/04/10	vie 02/04/10	39
41		Desconexión de jugadores	1 día?	lun 05/04/10	lun 05/04/10	40
42		**Desarrollo de aplicación cliente final en XNA**	**15 días?**	mié 31/03/10	**mar 20/04/10**	**7CC**
43		Cámara	2 días?	mié 31/03/10	jue 01/04/10	**7CC**
44		Detección de interacción del usuario y movimiento del jugador	1 día?	vie 02/04/10	vie 02/04/10	43

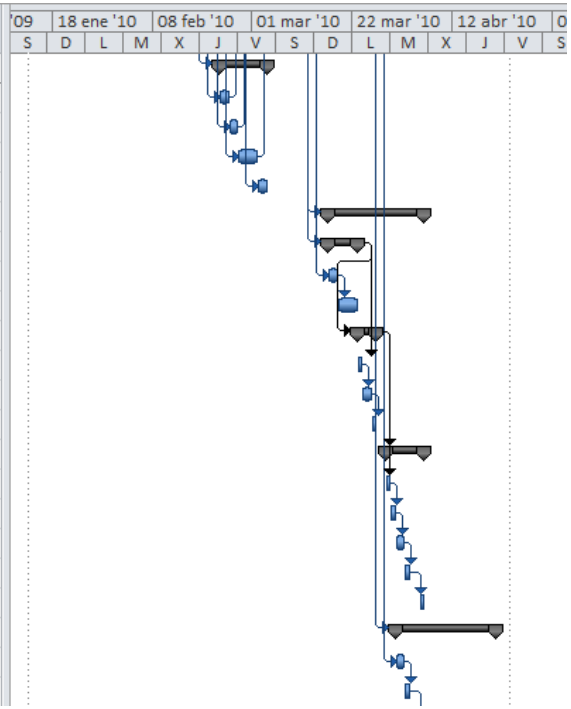

Imagen 50. Diagrama de Gantt - Planificación estimada 2/3

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
45		Carga de contenidos gráficos	4 días?	lun 05/04/10	jue 08/04/10	44
46		Representación gráfica de elementos en pantalla	2 días?	vie 09/04/10	lun 12/04/10	45
47		Desarrollo de sistema de colisiones	1 día?	mar 13/04/10	mar 13/04/10	46
48		**Inclusión de modo multijugador**	**5 días?**	**mié 14/04/10**	**mar 20/04/10**	**47**
49		Envío y rececpción de información del jugador al servidor	4 días?	mié 14/04/10	lun 19/04/10	47
50		Sistema de chat	1 día?	mar 20/04/10	mar 20/04/10	49

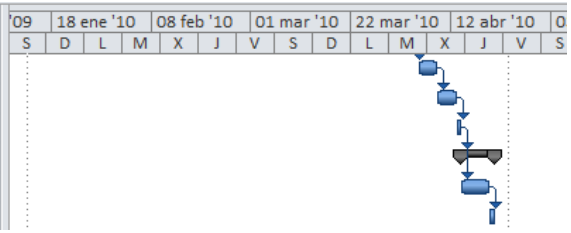

Imagen 51. Diagrama de Gantt - Planificación estimada 3/3

6.1.2.Planificación Gantt real

	Nombre de tarea	Duración	Comienzo	Fin	Predecesor
1	**Estudio de documentación**	145 días	mié 13/01/10	mar 03/08/10	
2	**Creación de memoria y documentación**	**145 días**	mié 13/01/10	mar 03/08/10	
3	Estado de la cuestión	24 días	mié 13/01/10	lun 15/02/10	
4	Análisis del problema	15 días	mar 16/02/10	lun 08/03/10	3
5	Solución del problema	35 días	mar 09/03/10	lun 26/04/10	4
6	Desarrollo de sistema de servidores	31 días	mar 27/04/10	mar 08/06/10	5
7	Desarrollo de aplicación XNA	35 días	mié 09/06/10	mar 27/07/10	6
8	Introducción	2 días	mié 28/07/10	jue 29/07/10	7
9	Apuntes finales	3 días	vie 30/07/10	mar 03/08/10	8
10	**Análisis del problema**	**10 días**	mar 16/02/10	**lun 01/03/10**	**4CC**
11	Impacto de diferentes tipos de juego	1 día	mar 16/02/10	mar 16/02/10	**4CC**
12	Sistemas de almacenamiento	1 día	mié 17/02/10	mié 17/02/10	11
13	Arquitecturas de red	3 días	jue 18/02/10	lun 22/02/10	12
14	Protocolos de comunicación de nivel de transporte	3 días	mar 23/02/10	jue 25/02/10	13
15	Concurrencia	2 días	vie 26/02/10	lun 01/03/10	14
16	**Diseño de solución**	**28 días**	mar 09/03/10	**jue 15/04/10**	**5CC**
17	Sistema de servidores y cliente	4 días	mar 09/03/10	vie 12/03/10	**5CC**
18	Secuenciación en el protocolo de comunicación	3 días	lun 15/03/10	mié 17/03/10	17
19	Modelo Cliente-Servidor UDP no concurrente	2 días	jue 18/03/10	vie 19/03/10	18
20	Modelo Cliente-Servidor híbrido TCP/UDP concurrente	3 días	jue 25/03/10	lun 29/03/10	24
21	Modelo Cliente-Servidor TCP no concurrente	2 días	jue 08/04/10	vie 09/04/10	25

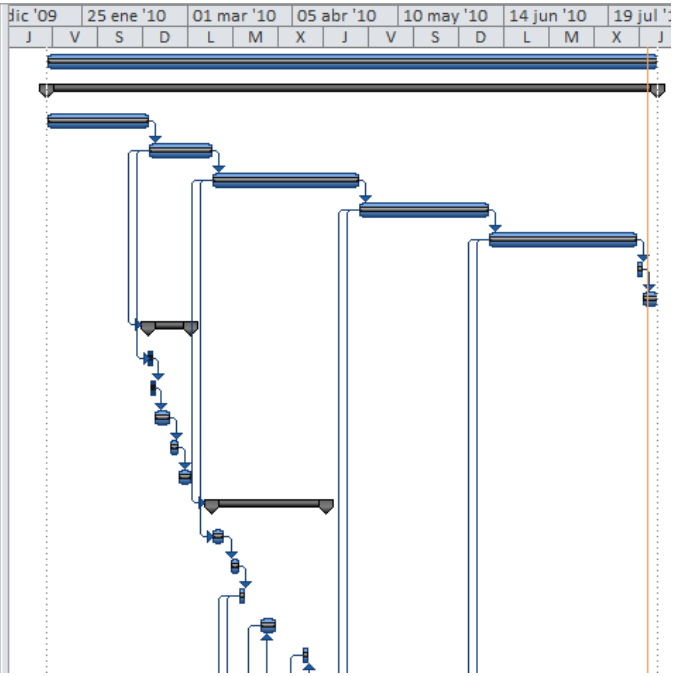

Imagen 52. Diagrama de Gantt - Planificación real 1/3

	Nombre de tarea	Duración	Comienzo	Fin	Predecesor
22	Modelo Cliente-Servidor TCP concurrente	3 días	mar 13/04/10	jue 15/04/10	26
23	**Implementación y pruebas de prototipos**	**23 días**	jue 18/03/10	lun 19/04/10	**19CC**
24	Modelo Cliente-Servidor UDP no concurrente	5 días	jue 18/03/10	mié 24/03/10	19CC
25	Modelo Cliente-Servidor híbrido TCP/UDP concurrente	10 días	jue 25/03/10	mié 07/04/10	20CC
26	Modelo Cliente-Servidor TCP no concurrente	3 días	jue 08/04/10	lun 12/04/10	21CC
27	Modelo Cliente-Servidor TCP concurrente	5 días	mar 13/04/10	lun 19/04/10	22CC
28	**Desarrollo de sistema de servidores final**	**25 días**	mar 27/04/10	**lun 31/05/10**	**6CC**
29	**Servidor de conexión**	**4 días**	mar 27/04/10	**vie 30/04/10**	**6CC**
30	Conexión y desconexión de servidores	2 días	mar 27/04/10	mié 28/04/10	**6CC**
31	Conexión de clientes	2 días	jue 29/04/10	vie 30/04/10	30
32	**Servidor de login**	**8 días**	**lun 03/05/10**	**mié 12/05/10**	**29**
33	Alta en el sistema de servidores	2 días	lun 03/05/10	mar 04/05/10	29
34	Comunicación con jugadores	4 días	mié 05/05/10	lun 10/05/10	33
35	Acceso a sistema de almacenamiento	2 días	mar 11/05/10	mié 12/05/10	34
36	**Servidor de juego**	**13 días**	**jue 13/05/10**	**lun 31/05/10**	**32**
37	Alta en el sistema de servidores	3 días	jue 13/05/10	lun 17/05/10	**32**
38	Conexión de jugadores	4 días	mar 18/05/10	vie 21/05/10	37
39	Broadcasting de información	2 días	lun 24/05/10	mar 25/05/10	38
40	Seguridad e integridad de la información transmitida	1 día	mié 26/05/10	mié 26/05/10	39
41	Desconexión de jugadores	3 días	jue 27/05/10	lun 31/05/10	40

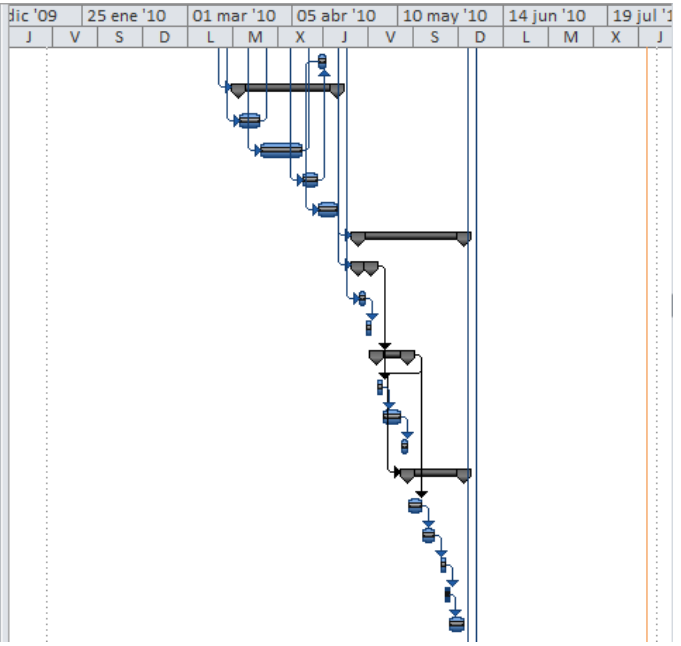

Imagen 53. Diagrama de Gantt - Planificación real 2/3

	Nombre de tarea	Duración	Comienzo	Fin	Predecesor
42	**Desarrollo de aplicación cliente final en XNA**	**28 días**	mié 09/06/10	**vie 16/07/10**	**7CC**
43	Cámara	3 días	mié 09/06/10	vie 11/06/10	**7CC**
44	Detección de interacción del usuario y movimiento del jugador	2 días	lun 14/06/10	mar 15/06/10	43
45	Carga de contenidos gráficos	2 días	mié 16/06/10	jue 17/06/10	44
46	Representación gráfica de elementos en pantalla	5 días	vie 18/06/10	jue 24/06/10	45
47	Desarrollo de sistema de colisiones	2 días	vie 25/06/10	lun 28/06/10	46
48	**Inclusión de modo multijugador**	**14 días**	**mar 29/06/10**	**vie 16/07/10**	**47**
49	Envío y rececpción de información del jugador al servidor	12 días	mar 29/06/10	mié 14/07/10	47
50	Sistema de chat	2 días	jue 15/07/10	vie 16/07/10	49

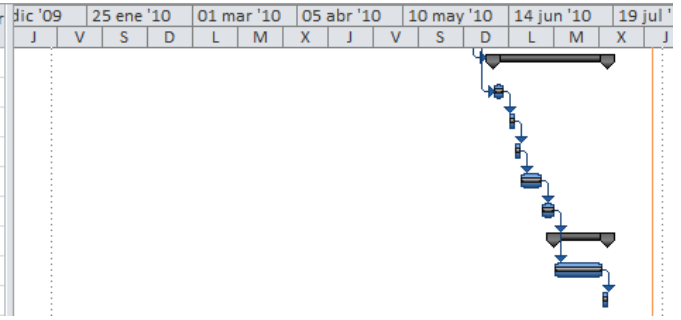

Imagen 54. Diagrama de Gantt - Planificación real 3/3

6.1.3.Análisis de planificación estimada y real

Como se puede observar a través de los diagramas Gantt, la duración final del proyecto (145 días) no coincide con la duración calculada para la planificación estimada(73 días), produciéndose una desviación que duplica lo estimado.

El motivo de la desviación producida se debe principalmente a dos causas:

- **Evolución del proyecto durante su desarrollo**, ya que partiendo de un objetivo original cerrado han aparecido nuevos retos sobre los que no se tenía intención de profundizar y finalmente han conformado la base del proyecto
- **Tiempo invertido en contrastar diferentes fuentes de documentación mayor del esperado**, ya que en muchas ocasiones la misma información no coincidía en múltiples referencias dando lugar a una mayor inversión de tiempo en búsqueda de soluciones que contrastaran los datos reflejados o incluso llegando a ser necesaria la realización de bancos de pruebas para obtener conclusiones.

Vistas las causas se puede ver la relación directa con la desviación presentada por la estimación y planificación presentada en los diagramas Gantt, ya que es en la fase de Diseño de la solución (debido a la necesidad de contrastar la información) y en la fase de Implementación (debido a la profundidad alcanzada en aspectos no esperados en el inicio del proyecto) donde mayor desviación se observa.

6.2.Presupuesto

Autor

Miguel Ángel García del Moral

Departamento

Departamento de informática

Descripción del Proyecto

- **Título.** Manual para el desarrollo de infraestructuras MMO
- **Duración.** 7 meses
- **Tasa de costes indirectos.** 20%

Presupuesto detallado del Proyecto

- **Personal**

Apellidos y nombre	N.I.F.	Categoría	Dedicación (hombres mes)	Coste hombre mes	Coste (Euro)
Peralta Donate, Juan	44445555-A	Ingeniero Senior	1	4.289,54	4.289,54
García del Moral, Miguel Ángel	99998888-B	Ingeniero	7	2.694,39	18.860,73
				Total	**23.150,27**

Tabla 11. Presupuesto detallado de personal

– **Equipos**

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable d)
Ordenador diseño gráfico	1.368,64	35	7	60	55,89
Ordenador programación	950,00	100	7	60	110,83
Servidor de datos	1.050,00	100	7	60	122,50
Disco duro externo	158,00	100	7	60	18,43
Multifunción Láser + Fax	179,00	100	7	60	20,88
				Total	**328,54**

Tabla 12. Presupuesto detallado de equipos

Los equipos que en el presupuesto se detallan son los siguientes:

- **Ordenador para diseño gráfico**. Ordenador utilizado para el diseño del contenido gráfico, sus especificaciones técnicas son:

  - **Caja.** Antec P183
  - **Procesador**. Intel Core i5 750 2.66Ghz Box Socket 1156
  - **Placa Base**. Gigabyte GA-P55A-UD3 Socket 1156
  - **Memoria RAM**. 4GB DDR3 Mushkin BlackLine 10666 DDR3 1333 4GB 2x2GB CL7
  - **Disipador**. Cooler Master Hyper 212 Plus
  - **Fuente de Alimentación**. Antec Basiq Power 550W Modular.
  - **Disco duro**. Western Digital Caviar Black 1TB SATA3 Maestro
  - **Tarjeta Gráfica**. Gigabyte GeForce GTX470 1280MB GDDR5
  - **Grabadora de DVD**. LG GH22NS40 Grabadora DVD SATA 22X Negra
  - **Tarjeta de sonido**. Realtek ALC889A High Definition Audio 2/4/5.1/7.1-channel

- **Ordenador para desarrollo**. Ordenador utilizado para el desarrollo tanto de la documentación generada por el proyecto como para el desarrollo de la prueba de concepto, sus características técnicas son:

    - **Caja.** Antec Three Hundred
    - **Fuente de alimentación.** Anteq Basiq Plus 550W Modular
    - **Procesador.** Intel Core i7 930 2.80Ghz Box Socket 1366
    - **Placa Base.** Asus P6T SE
    - **Memoria RAM.** 6GB Mushkin SilverLine PC3-12800 DDR3 1600 3x2GB CL9
    - **Disco Duro.** Western Digital Caviar Black 1TB SATA3 Maestro
    - **VGA.** Sapphire Radeon HD 5770 1GB GDDR5
    - **Grabadora de DVD.** LG SATA Negra
    - **Tarjeta de sonido.** Realtek ALC889A High Definition Audio 2/4/5.1/7.1-channel

- **Servidor de datos**. Equipo utilizado para proveer un sistema de ficheros compartidos, gestionar el dominio de la empresa y gestión de control de versiones del proyecto. Sus especificaciones técnicas son:

    - **Caja.** Cooler Master CM 690 Pure Black
    - **Fuente de alimentación.** Anteq Basiq Plus 550W Modular
    - **Procesador.** Intel Core i7 930 2.80Ghz Box Socket 1366
    - **Disipación y ventilación del procesador**. Scythe Ninja 3
    - **Placa Base.** Asus P6T SE
    - **Memoria RAM.** 6GB Mushkin SilverLine PC3-12800 DDR3 1600 3x2GB CL9
    - **Disco Duro.** 2 x Seagate Barracuda LP 2TB SATA2 32MB MAESTRO
    - **VGA.** Asus Radeon HD 4350 Silent 512MB GDDR2 LP
    - **Tarjeta de sonido.** Realtek ALC889A High Definition Audio 2/4/5.1/7.1-channel

- **Disco duro externo.** Disco duro para conectarlo por USB para sacar de la empresa las copias de seguridad del contenido generado por el Proyecto, sus especificaciones son:

    - **Modelo.** WD My Book Mirror Edition 2TB 3.5" Disco Duro USB
    - **Capacidad.** 2 TB de almacenamiento de datos
    - En su interior contiene dos discos duros de 2TB cada uno montados en RAID1 para ofrecer una mayor integridad de la información almacenada
    - Posibilidad de reemplazar los discos duros que contiene para ampliar la capacidad o sustituir un disco que presente fallos

- **Multifunción, láser y fax**. Utilizada para la impresión de documentación y facilitar el desarrollo del Proyecto, sus especificaciones son:

    - **Modelo**. Brother MFC-7320

– **Otros costes directos**

Descripción	Empresa	Costes imputable
Fungible	Ofistore	200,00
Viajes	-	140,00
Licencia Microsoft Windows Server 2008	Arinza	526,92
2 x Licencia Windows 7 Ultimate	PC-Componentes	364,00
Licencia Microsoft Source Safe 2005	Microsoft Store	396,00
Licencia Backup Exec	Symantec Store	3.120,00
Licencia Microsoft Office Hogar y PYME	Microsoft Store	379,00
	Total	**5.215, 92**

Tabla 13. Presupuesto detallado de otros costes directos

- **Bienes fungibles**. Aquí se incluye el material de oficina:

- **Viajes**. Costes de transporte para realizar reuniones de seguimiento con el equipo de trabajo.

- **Licencias**. Licencias de software necesarias para los equipos y el desarrollo del Proyecto.

Presupuesto total del Proyecto

- **Personal.** 23.150 €
- **Coste imputable en amortización equipos informáticos.** 329 €
- **Costes de funcionamiento.** 5.216 €
- **Costes indirectos**. 5.739 €
- **Total.** 34.434 €

6.3.Comercialización del proyecto

Pese a que el objetivo de este proyecto no consiste en la creación de un videojuego multijugador masivo online, a continuación se identifican posibles puntos de comercialización que garantizarían rentabilidad al desarrollo de este género de videojuego:

- **Cuota mensual**, es un sistema muy extendido en este género que garantiza un ingreso continuado y que serviría de espina dorsal para los costes introducidos por el mantenimiento de los servidores que alojen las aplicaciones servidoras del juego.
- **Microtransacciones**, este sistema cada vez toma mayor presencia en el género, consiste en ofrecer elementos exclusivos a los jugadores que de otro modo no podrían conseguir, pueden ser desde cambios en el nombre del personaje hasta ropa diferente a la del resto de jugadores.
- **DLC**, al igual que las microtransacciones este sistema está viviendo un auge gracias a la integración de las comunicaciones en la red con el mundo del videojuego, a través de él y previo pago por parte del jugador, puede ampliar las características del juego introduciendo nuevas misiones, personajes, etc. Se diferencia de las microtransacciones en que los cambios aportados son más notables, llegando a ofrecer posibilidades que antes ni existían.
- **Publicidad**, insertada en banners en la página del videojuego o dentro del propio mundo recreado por el videojuego.

7.Conclusiones

La evolución de los objetivos durante el desarrollo del proyecto influyó en la complejidad de éste por dos motivos: en primer lugar porque el diseño del sistema establecido para la idea original no era compatible con la idea final ya que los objetivos cubrían ámbitos diferentes, lo que supuso la necesidad de reestructuración del planteamiento; y en segundo lugar porque en el desarrollo de sistemas de juego multijugadores masivo en línea, pese a haber mucha documentación a través de foros y demás portales Web, no se puede decir que haya un verdadero conocimiento por parte de estas comunidades ya que se fundamentan en teorías que llevadas a la práctica podrían no dar el resultado esperado si no se ha dimensionado bien el problema.

Respecto a la lectura final que se puede realizar del estudio del problema planteado, se concluye la relevancia de cada una de las decisiones tomadas, y como unas decisiones afectan al resto de ámbitos en el diseño del sistema final siendo cada decisión crucial para un funcionamiento correcto, como por ejemplo el efecto que tendrá en el protocolo que seguirán los distintos nodos de la red de clientes (jugadores) y servidores en el caso de decantarse por una arquitectura de red u otra, obligando al diseñador a establecer un paso de mensajes diferente en diferentes arquitecturas, que pueden afectar al protocolo de comunicaciones elegido y éste a la información que la aplicación enviará, etc.

Por ello para no encontrarse con limitaciones y tener la mayor capacidad de decisión se afrontó el problema desde la plataforma de desarrollo XNA en la cual el límite está impuesto por su librería y la plataforma .NET la cual permite a los diseñadores y desarrolladores realizar el trabajo poniendo a su alcance todas las herramientas necesarias para implementar la solución deseada.

Por otro lado, se concluye que para alcanzar con éxito el fin en la elaboración de un proyecto de esta magnitud es necesario contar con un equipo de desarrollo con decenas de personas entre los que se tienen que encontrar expertos en los ámbitos cubiertos durante todo el proceso seguido en el proyecto, desde personas encargadas a la gestión de proyecto, hasta los programadores con experiencia en la plataforma .NET y XNA, pasando por diseñadores de sistemas, expertos en telecomunicaciones que dimensionen adecuadamente el problema de la transmisión de datos y definan los protocolos que serán utilizados, diseñadores 2D y 3D para el arte gráfico del juego, músicos para la composición musical y la generación de efectos de sonido, gente con experiencia comercial para la promoción del producto final, etc.

Cabe destacar también el interés por las comunidades en este tipo de proyectos, ya que durante el desarrollo inicial del proyecto se buscaron colaboradores en el foro de **Desarrollo de videojuegos** en el portal Web www.meristation.com para una vez finalizada la creación de la infraestructura que diera soporte al MMO, continuar con el desarrollo del juego, y se obtuvieron 7249 visitas con 60 participaciones de usuarios en el hilo dedicado al tema, de lo que se puede concluir de comparar los resultados con otros hilos de conversación del foro el

alto grado de interés de la gente por el tema dado el alto número de visitas, y por otro lado el desconocimiento del sector del MMO dada la poca participación.

En la siguiente tabla se muestran los datos reflejados por los hilos de mayor impacto dedicados al desarrollo de videojuegos en este foro:

Nombre del hilo	Número de visitas	Mensajes escritos
Ectoplasmic Wars	8686	103
Desarrollo de videojuego [XNA]	7249	60
Nemesis Online	5503	21
Avatar Insane Race	3344	76
Navy Wars	2203	26

Tabla 14. Interés por el desarrollo de un proyecto XNA

8.Líneas futuras

Líneas futuras de desarrollo respecto a la infraestructura MMO y a su integración en otros proyectos.

A continuación se exponen las líneas futuras de desarrollo que se pueden seguir en el proyecto presentado para convertirlo en un videojuego completo y que por falta de tiempo y tratar aspecto no marcados en el objetivo final del proyecto no han sido incluidas:

- **Bases de datos**

Diseño y uso de bases de datos para almacenar los datos de los personajes manejados por los usuarios para facilitar la introducción de nuevos campos en futuras ampliaciones de funcionalidad del videojuego.

- **Diseño de personajes**

Incluir un diseñador de personajes para permitir que los jugadores puedan crear su avatar personal escogiendo características físicas como el sexo, estatura, complexión, detalles faciales…, ya que en videojuegos donde multitud de jugadores van a compartir una experiencia jugable es importante ofrecer una herramienta que les permita diferenciarse del resto de jugadores a través de la personalización de su alter ego.

- **Diseño de niveles**

Ampliar y desarrollar distintos niveles para que los jugadores tengan distintas localidades en las que interactuar, incluso sería interesante ampliar la funcionalidad del videojuego dando la posibilidad a los jugadores de crear sus propios niveles para compartirlos con el resto.

- **Incluir opciones que refuercen la jugabilidad**

Ofrecer un mayor abanico de opciones jugables a través de la interacción de los usuarios con otros usuarios y con el entorno.

- **Mejora de la interfaz gráfica**

Conforme se amplíen las opciones de jugabilidad habrá que rediseñar la interfaz gráfica mostrada para informar al jugador de las posibles acciones que puede realizar.

- **Diseño de sonido**

La versión actual carece de efectos de sonido y música de ambiente, es importante incluir elementos para intensificar momentos concretos y acompañar la experiencia jugable.

- **Sistema de lista de amigos**

Al tratarse en este manual el desarrollo de un videojuego multijugador masivo, otra funcionalidad adicional que sería interesante incluir es la capacidad de guardar una relación de los nombres de otros usuarios que sean amigos de éste para saber si están conectados en un determinado momento, su localización, ofrecer la posibilidad de mantener una conversación privada con ellos, etc. De igual modo también sería interesante guardar una relación de usuarios que al jugador no le interese interactuar con ellos, como si se tratara de una lista negra.

- **Optimización de recursos**

En la codificación actual se carga en memoria todos los elementos del mundo incluso aunque estos no estén dentro del campo visual del jugador, de modo que de tener un mapeado de mayor tamaño se estarían empleando recursos del equipo para su procesamiento.

Esto mismo se aplica a los datos enviados por los jugadores, sería interesante controlar que jugadores están a una determinada distancia del jugador local para no enviarle información que no le afecte y así ahorrar ancho de banda.

- **Encriptar la información transmitida**

Como se explicó en el apartado 4.2.3 Arquitectura de red, seguridad, en el momento en que sea necesario transmitir datos reales del usuario o realizar determinadas acciones en el sistema se tendrá que realizar haciendo uso de un sistema de cifrado para garantizar la autenticación, confidencialidad, no repudio e integridad en la transmisión de los datos.

- **Ajuste de coordenadas por latencia**

Debido a las dificultades que introduce el realizar un ajuste correcto de las coordenadas de posición de modo que todos los jugadores tengan la misma información en el mismo momento, queda pendiente el desarrollo de una implementación más compleja.

9.Bibliografía

API
http://en.wikipedia.org/wiki/Application_programming_interface
http://www.webopedia.com/TERM/A/API.html
http://compnetworking.about.com/od/softwareapplicationstools/g/bldef_api.htm

Arquitectura de red modelo cliente-servidor
http://es.wikipedia.org/wiki/Cliente-servidor
http://www.csae.map.es/csi/silice/Global71.html
http://www.monografias.com/trabajos24/arquitectura-cliente-servidor/arquitectura-cliente-servidor.shtml

Arquitectura de red P2P
http://es.wikipedia.org/wiki/Peer-to-peer

Arquitecturas de redes
http://html.rincondelvago.com/arquitectura-de-redes.html

Cifrado asimétrico
http://es.wikipedia.org/wiki/Criptograf%C3%ADa_asim%C3%A9trica

Cifrado simétrico
http://es.wikipedia.org/wiki/Criptograf%C3%ADa_sim%C3%A9trica

Concurrencia
http://www.alegsa.com.ar/Dic/concurrencia.php

Creación de un MMORPG
http://www.devmaster.net/articles/building-mmorpg/

ENET
http://enet.bespin.org/

Game Maker
http://www.yoyogames.com/make
http://es.wikipedia.org/wiki/Game_Maker

IDE
http://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

Introducción a la programación de juegos multijugador
http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming

Irrlicht
http://irrlicht.sourceforge.net/features.html
http://en.wikipedia.org/wiki/Irrlicht_Engine

Motor de juego
http://es.wikipedia.org/wiki/Motor_de_videojuego
http://en.wikipedia.org/wiki/Game_engine
http://www.devmaster.net/engines/

OGRE3D
http://www.ogre3d.org/about/features

Patrón Composite
http://es.wikipedia.org/wiki/Composite_%28patr%C3%B3n_de_dise%C3%B1o%29

Patrón Observer
http://es.wikipedia.org/wiki/Observer_%28patr%C3%B3n_de_dise%C3%B1o%29

Patrones de diseño
Patrones de diseño – *Erich Gamma*
Editorial Adisson Wesley

PNG – Portable Network Graphics
http://www.libpng.org/pub/png/

Programación de multijugador y redes
http://www.gamedev.net/community/forums/showfaq.asp?forum_id=15

SDL – Simply DirectMedia Layer
http://es.wikipedia.org/wiki/Simple_DirectMedia_Layer
http://www.libsdl.org/

Shader
http://www.neoteo.com/pixel-shaders-y-vertex-shaders.neo
http://es.wikipedia.org/wiki/Shader

Sistemas distribuidos
http://es.wikipedia.org/wiki/Computaci%C3%B3n_distribuida
http://www.monografias.com/trabajos16/sistemas-distribuidos/sistemas-distribuidos.shtml

TCP
http://es.wikipedia.org/wiki/Transmission_Control_Protocol

TCP RTF-793
http://www.ietf.org/rfc/rfc793.txt

Torque Game Builder
http://www.torquepowered.com/products/torque-2d
http://en.wikipedia.org/wiki/Torque_Game_Engine

UDP
http://es.wikipedia.org/wiki/UDP

UDP RTF-768
http://www.ietf.org/rfc/rfc768.txt

Unreal Engine
http://www.unrealtechnology.com/

XNA
http://msdn.microsoft.com/en-us/aa937791.aspx
http://es.wikipedia.org/wiki/Microsoft_XNA
http://creators.xna.com/es-ES/

10.Anexos

10.1.Anexo I: Glosario de términos

A continuación se detallan los términos, acrónimos y definiciones que serán utilizados para el desarrollo del presente documento:

ACRÓNIMO	DEFINICIÓN
2D	Dos dimensiones
3D	Tres dimensiones
Adaptador de red o NIC	*Network Card Interface*, elemento hardware que se necesita para comunicar máquinas en una red. El trabajo del adaptador de red es el de convertir las señales eléctricas que viajan por el cable o las ondas de radio en una señal que pueda interpretar el ordenador.
API	*Application Programming Interface*. Es el conjunto de funciones, procedimientos, clases, métodos (en función del paradigma de programación) que ofrece una biblioteca para ser utilizado en el desarrollo de aplicaciones.
ARP	*Address Resolution Protocol*, protocolo de nivel de red en el modelo OSI. Se encarga de obtener la dirección física (MAC) de un equipo partiendo de la dirección IP.
ASN.1	*Abstract Syntax Notation One*, protocolo de nivel de presentación en el modelo OSI cuya función consiste en representar los datos con independencia de las máquinas de origen y destino en caso de que su representación interna sea diferente.
Broadcast	Transmisión de datos donde un nodo envía información a una multitud de nodos sin necesidad de reproducir la misma transmisión nodo por nodo.
Cliente	En comunicaciones, es la máquina en la que se ejecuta la aplicación que envía peticiones a una máquina en la que se ejecuta la aplicación servidora, y que por regla general esperará una respuesta por parte del servidor una vez realizada la petición.
Datagrama	Es el nombre que reciben los mensajes (información enviada a través de la red) cuando se usa el protocolo UDP.
Dirección IP	Conjunto de números que identifican una máquina en la red. En la versión usada actualmente (IPv4) se compone de 4 grupos de números que van del 0 al 255 y separados por un punto, siendo un ejemplo 57.57.100.202
Ethernet	Es un estándar de redes de computadoras de área local con acceso al medio por contienda CSMA/CD, actúa en la capa de red del modelo TCP/IP.

Tabla 15. Glosario de términos

FPS	Aplicado a los videojuegos tiene dos acepciones: - First Person Shooter. Equivale a Shooter que ya fue definido en este apartado. - Frames per second. Frames por segundo, medida utilizada para comprobar el rendimiento de un videojuego contando el número de imágenes de una sucesión de imágenes que se han producido en un segundo de juego.
Frame	En relación a los videojuegos, es una imagen dentro de una serie de ellas. Se utiliza para comprobar el rendimiento, midiendo la velocidad de imágenes que se están viendo en pantalla en un segundo (frames por segundo).
Framebuffer	Se conoce con este nombre a la categoría de dispositivos gráficos que basan su funcionamiento en la representación de cada uno de los píxeles de la pantalla como posiciones de memoria en la memoria principal (RAM) del equipo.
GPU	*Graphics Processing Unit*. Es el procesador cuyo objetivo consiste en procesar la lógica asociada al manejo de gráficos en 2 y 3 dimensiones con el fin de liberar carga al procesador central. Adicionalmente, las GPUs suelen estar diseñadas con la inclusión de ciertas operaciones gráficas de uso típico que denominan primitivas para mejorar el rendimiento en el trabajo más habitual.
HUB	Dispositivo que permite centralizar el cableado de una red y poder ampliarla siendo capaz de retransmitir una señal recibida, de modo que la repetirá por los diferentes puertos de entrada.
ICMP	*Internet Control Message Protocol*, protocolo de nivel de red en el modelo OSI. Su función es controlar y notificar errores en la del protocolo IP.
IDE	*Integrated Development Environment*. Son programas compuestos por un conjunto de herramientas para dar soporte a la programación en uno o más lenguajes de programación.
IP	*Internet Protocol,* Protocolo de internet, protocolo de nivel de red en el modelo OSI cuya función consiste en transmitir datos de una máquina origen a una destino.
Latencia	En comunicaciones, se denomina latencia a la suma de los retardos producidos por la red (demora en propagación y tiempo de transmisión por la red)
MAC	*Media Access Control*, control de acceso al medio. Es la subcapa de la capa de Enlace del modelo OSI encargada de controlar el acceso al medio de transmisión por parte de diferentes dispositivos que comparten el mismo medio, detectar y corregir errores en datos transmitidos, descartar tramas duplicadas, etc.
MMO	*Massive Multiplayer Online*, multijugador online masivo. Estilo de juego en el que jugadores de forma masiva están conectados y comparten una misma experiencia jugable.
Mutex	Mecanismo de sincronización para garantizar un acceso seguro a un recurso compartido en programación concurrente.

NetBIOS	Network Basic Input/Output System, protocolo de nivel de sesión en el modelo OSI. Permite la comunicación del software de un sistema operativo de red con un hardware específico.
OSI	*Open System Interconnection*, modelo de referencia de Interconexión de Sistemas Abiertos definido por la Organización Internacional de Estandarización para establecer un marco de referencia en la definición de arquitecturas de interconexión de sistemas de comunicaciones.
P2P	*Peer to peer*, de par a par. Es una red en la que de forma teórica todas las máquinas se comportan del mismo modo y ejercen como clientes y servidor, en la práctica según el tipo de implementación puede ser necesario contar algún nodo en la red que cumpla un rol específico.
Paquetes	Es el nombre que reciben los mensajes (información enviada a través de la red) cuando se usa el protocolo TCP.
Pixel	Un píxel o pixel (acrónimo del inglés picture element, "elemento de imagen") es la menor unidad homogénea en color que forma parte de una imagen digital. Las imágenes se forman como una matriz rectangular de píxeles, donde cada píxel forma un área relativamente pequeña respecto a la imagen total.
Puerto	En comunicaciones, es un número del 0 al 65535 (2^16) que sirve para identificar por donde se va a realizar la comunicación en protocolos de transporte TCP/IP.
RARP	*Reverse Address Resolution Protocol*, protocolo de nivel de red en el modelo OSI. Realiza el proceso inverso a ARP, se encarga de obtener la dirección IP a través de la dirección física (MAC)
RCP	*Remote Procedure Call*, protocolo de nivel de sesión en el modelo OSI. Permite la ejecución de código de un programa en una máquina remota sin necesidad de que el desarrollador tenga que prestar atención a las comunicaciones entre ambas máquinas.
RPG	*Role Playing Game*, Juego de rol. Estilo de juego en que el jugador toma el control de un personaje o personajes que evolucionarán durante la experiencia jugable.
Sandbox	Modo de juego no lineal que presenta al jugador desafíos que pueden ser completados en un número de secuencias diferentes o libres, concediendo mucha más libertad de juego, en contraposición con un modo de juego lineal que presentará una serie de retos fija, a modo de historia.
Servidor	Máquina en la que se ejecuta la aplicación que recibe las peticiones de los clientes y que por regla general produce una respuesta que devolverá al cliente.
Shader	Es una tecnología proporcionada a nivel de hardware por la tarjeta gráfica y destinada a proporcionar al programador una interacción con la GPU que permite realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla sobre los gráficos presentados en pantalla.
Shooter	Estilo de videojuegos de disparos en el que el jugador toma el control de un personaje armado con el que interactuará con el entorno disparando de forma voluntaria.

Socket	Elemento abstracto a través del cual se comunican las aplicaciones. Se compone de la dirección IP de la máquina y el puerto por el que se va a comunicar.
Speed-hack	Técnica realizada en juegos para incrementar la velocidad de movimiento del jugador de forma ilegal.
Sprite	Se denomina sprite a una serie de imágenes utilizadas para animar gráfiso en 2 dimensiones, unidas en un mismo archivo una al lado de otra y que representan al mismo objeto o personaje en distintas posiciones para posteriormente realizar su animación. Esta forma de representar gráficos resulta más eficiente ya que el tiempo de carga de una imagen es muy superior al que ocupa seleccionar y mostrar un área de una imagen ya cargada en memoria.
SSL	*Secure Sockets Layer*, protocolo de nivel de sesión en el modelo OSI. Es un protocolo criptográfico que proporciona una comunicación segura por la red.
TCP	*Transmisión Control Protocol*, Protocolo de control de transmisión. Protocolo de nivel de transporte en el modelo OSI cuyas principales características son que es orientado a conexión y que garantiza la entrega de la información transmitida.
Thread	Hilo, se refiere a la programación concurrente usando procesos ligeros.
Tileset	Conjunto de fragmentos de imágenes denominados tile. Un tile es una parte seleccionado de un sprite.
UDP	*User Datagram Protocolo*, Protocolo de datagramas de usuario. Protocolo de nivel de transporte en el modelo OSI, sus principales características son que no es orientado a conexión, no tiene confirmación ni control de flujo.
X.25	Es un estándar para redes de área amplia de conmutación de paquetes, actúa en la capa de red del modelo TCP/IP.
XML	*Extensible Markup Language*, Lenguaje de marcas extensible. Permite la definición de gramáticas de lenguajes específicos.

[1] http://en.wikipedia.org/wiki/William_Higinbotham
[2] http://www.atari.com/
[3] http://www.nintendo.es/
[4] http://www.sega.es/
[5] http://www.sony.es/section/home
[6] Programmed Logic for Automated Teaching Operations
http://en.wikipedia.org/wiki/PLATO_(computer_system), http://platohistory.org/
[7] http://en.wikipedia.org/wiki/Bulletin_board_system, http://www.bbscorner.com/
[8] Neal Stephenson. Snow Crash. 1992
[9] http://es.wikipedia.org/wiki/Computaci%C3%B3n_distribuida,
http://www.dei.uc.edu.py/tai2002/SD/discom.htm
[10] http://es.wikipedia.org/wiki/Tolerancia_a_fallos
[11] ISO - International Organization for Standardization. http://www.iso.org/iso/home.html
[12] User Datagram Protocol. http://www.faqs.org/rfcs/rfc768.html
[13] Sitio web del proyecto: http://enet.bespin.org/
[14] Defense Advanced Research Project Agency. http://www.darpa.mil/
[15] http://www.copyright.es/
[16] http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx
[17] http://www.opengl.org/
[18] Sitio web del desarrollador del juego: http://ng.neocron.com/
[19] Sitio web del desarrollador del juego: http://www.mag.com/gate.html
[20] Sitio web del desarrollador del juego: http://www.online-multiplayer.com/huxley/
[21] http://drift.ijji.com/
[22] http://www.trackmania.com/
[23] http://www.dark-wind.com/
[24] http://projectpowder.outspark.com/
[25] http://freestyle.gamekiss.com/main.jce
[26] http://fantasytennis.es.alaplaya.net/
[27] http://www.travian.net/
[28] http://www.beyondprotocol.com/
[29] http://ultracorps.sjgames.com/
[30] http://www.wow-europe.com/es/index.xml
[31] http://www.cityofheroes.com/
[32] http://iro.ragnarokonline.com/
[33] http://secondlife.com/
[34] http://us.playstation.com/psn/playstation-home/
[35] Radu Privantu, Jefe de proyecto de Eternal Lands. http://www.devmaster.net/articles/building-mmorpg/ . 06/08/2004
[36] http://www.microsoft.com/downloads/en/details.aspx?FamilyID=80782277-d584-42d2-8024-893fcd9d3e82&displaylang=en
[37] Real Academia de la Lengua Española. Vigésima segunda edición. http://www.rae.es/rae.html
[38] Wikipedia. http://es.wikipedia.org/wiki/Antih%C3%A9roe
[39] http://www.microsoft.com/es/es/default.aspx
[40] http://www.xna.com/